# Multics Emacs: The History, Design and Implementation



**April 8, 1996**

What follows is my vast, unpublished 1979 "Mother of All Multics Emacs papers" from which all of my lesser and greater Emacs papers, published, internal, and unpublished, were ultimately excerpted, including *Prose and CONS: A Commercial Text-Processing System in Lisp* in the 1980 Lisp Conference proceedings and the 1980 Honeywell conference paper. It's about time to expose it, and the WWW/HTML is now the ideal vehicle.

Multics is no longer produced or offered for sale; Honeywell no longer even makes computers. People edit on computers on their desktop so cheap and fast that not only do redisplay algorithms no longer matter, but the whole idea of autonomous redisplay in a display editor is no longer a given (although autonomous redisplay's illustrious child, WYSIWYG, is now the standard paradigm of the industry.). There is now no other kind of editor besides what we then called the "video editor". Thus, all of the battles, acrimony, and invidious or arrogant comparisons in what follows are finished and done with, and to be viewed in the context of 1979 -- this is a *historical* document about Multics and the evolution of an editor. It is part of the histories of Multics, of Emacs, and of Lisp.

Many of the deficiencies of Multics described here were thereafter remedied, not only by Emacs, but by a managed-video system inspired by Emacs. Although it started out as rebellious "hack", Multics Emacs became an integral part of the Multics product.

The term ARPANET refers to the early stages of what is now called the **INTERNET**. **ARPA** was the Advanced Research Projects Agency of the U.S. Department of Defense, who instigated and underwrote its development.

Please enjoy.

---

## Table of Contents

---



---

# Multics Emacs: The History, Design and Implementation

## Bernard S. Greenberg -- 15 August 1979

(Paper on Multics Emacs, intended audience unclear. Idea is to put this stuff down on "paper" for posterity.) Brackets ([xxx]) denote references to the Bibliography at the end of the main text.

Multics Emacs is a video-oriented text preparation and editing system running on Honeywell's Multics [Multics] system, being distributed as an experimental offering in Multics Release 7.0. From the viewpoint of Multics, it represents the first video-management software to be implemented, the first time character-at-a-time-interaction has been used, and a radical and complete departure from other editing and text preparation tools and techniques prevalent on Multics. From the viewpoint of similar systems running elsewhere, several features are noteworthy, including a major commitment to the programming language Lisp [Moonual] [Chineual], user-accessible extensibility through Lisp, and an internal implementation designed from the start (as is often not the case) with display-oriented editing in mind. The seemingly innate expense of video-oriented interaction has also led to the development of performance enhancement techniques applicable to any such system. The growth of Multics Emacs and its user community on MIT's Multics system has also led to the development of protocols on for the ARPANET [Arpanet] designed to facilitate the use of video-oriented software in a device-independent fashion.

Multics Emacs is currently in widespread use at three Multics exposure/development sites, serving about 60 regular users. Due to the lack of previous video-oriented software on Multics, not many users have a large number of high-speed video terminals connected to their Multics system. Thus, much usage of Multics Emacs is via 300- and 1200-baud dialup lines. This fact, combined with the acknowledged expense and resource consumption of Multics Emacs, places Multics Emacs among a choice of editing tools: given current resource economies and communications devices at current sites, it is not always the tool of choice, even among its most fervid partisans.

This paper describes the background and history of Multics Emacs, the previous developments, and the climate in which it was created. As many of the salient features and design and implementation criteria and decisions as possible are stated, as well as experience with the design and implementation. Where

important, complete algorithms are detailed, notably the *redisplay algorithm*, detailed in the Appendix.

# I. Prehistory of Editing on Multics

Four editors were in common use on Multics before the introduction of Multics Emacs. They are still in widespread use. Two of them, "edm" [MPM] and "qedx" [QUG], are standard, and intended as end-user interfaces. Both of these editors are line-oriented, printing-terminal editors of a conventional mold. When in *command mode*, lines full of editor commands are typed to effect the positioning to, printing, deleting, or changing (by string substitution) of lines, or entry into *input mode*, which is the only way new text may be entered. The typing of a reserved character sequence exits input mode back to command mode. These editors maintain buffers (edm maintains one text buffer, qedx many) as character strings with a gap in the middle, which represents the current `point` at which text may be inserted. Neither editor can address individual characters. The edm editor is extremely limited, is not programmable, and intended for novice users. It was developed from the EDL [CTSS] editor interface on CTSS. The qedx editor was developed as a stripped-down (for runtime efficiency) version of the QED editor [CTSS] [Bell QED], also on CTSS. (Many of the original Multics developers had worked on CTSS). Both editors are oriented towards a low-speed, half-duplex, printing terminal interface, where minimal typeout is a design criterion. It was solely with these editors that the bulk of the Multics operating system and applications were entered, edited, and maintained.

An implementation of TECO (a version of which is the standard DEC editor) was built on Multics in 1971. This version was derived from the original TECO implementation, on the MIT AI Lab's PDP-10 [TECDOC]. Unlike the latter, Multics TECO has no display support, and does not have the complex control-structure constructs of the AI Lab editor. Multics TECO is supported by Honeywell as a tool, which is to say, not an end-user interface [Tools manual]. Multics TECO has had a handful of supporters, but has never really achieved widespread use. TECO presents as an interface a repertoire of single-character commands designed to provide substantial power as a text-processing programming language. While trying to succeed as both an editing language and a programming language, TECO falls short at both due to the necessary compromises. Nevertheless, a principal feature of TECO is the ability to construct powerful text-processing programs (*macros*) readily, and Multics TECO has often been used for such.

The remaining editor which is widely used on Multics is a version of qedx ("ted") developed by James A. Falksen, adding substantial power, including the ability to address characters and many commands which qedx is lacking. Many fail-soft features and help features also stand out in Falksen's editor. Due to a large variety of compatibility constraints and product schedules, no improvements to qedx have been made in the last five years, resulting in widespread dissatisfaction among the Multics user community with the inadequacies of qedx. Thus, Falksen's editor, which is not a Honeywell product, or distributed with Multics, has achieved almost exclusive use at many Multics sites. It is not conceptually different from qedx, differing principally in the vastly extended command repertoire. There is no published documentation available.

# II. History of Video-oriented Editing on ITS

The impetus for Multics Emacs came from the outside. In late 1977 and early 1978, the author became acquainted with Emacs [Stallman] on the ITS [ITSDOC] system at the MIT AI Lab. ITS Emacs grew out of ITS TECO during the period 1975-1977. Some background on the history and development of

ITS Emacs is now in order.

ITS TECO was designed for use in a display-oriented environment. In its normal mode of usage, before the evolution of Emacs, it split the user's screen into two regions, one in which the user typed editor commands and one in which a region of the text being edited was displayed. As the user completed a sequence of TECO commands, the editor would update the image of the text buffer being displayed on the screen. While obviating the need for a "*print* command" (which shows lines of a buffer on request), this technology still revolved around a user typing a line of "editor commands," and "activating" them. Included therein are requests to insert text, as normal editor commands. An appropriate section of the buffer to display would be chosen automatically.

The next advance in editor technology on ITS was the development of *Control R*, or *Real-time edit* mode in TECO. This mode (so called because the "Control R" character invoked the command which entered it) allowed a selected set of characters to be used as "real-time" commands: instead of entering a string of editor commands, a user would type one of these characters (such as "Control D" for "delete a character") and the effect would immediately be made visible in the buffer display. As each command character was typed, the text displayed would be appropriately modified. All of the command characters, a fixed, small set, were drawn from the "Control" characters of the ASCII character set, i.e., non-printing characters. The typing of printing characters caused them to be inserted directly into the buffer (and appear on the screen). TECO maintains a *point*, a virtual pointer, into its text buffer: in Control R mode, an identification of TECO's *point* with a terminal's cursor is made, so that modifications to the text being edited appear "at the cursor" in the buffer display.

The novelty of Control R mode was the freedom from editor requests: the terminal became an "editing device," upon which certain keys had predictable effects upon the text displayed thereupon, as a "machine," as opposed to an "interactive editor." In fact, what had been achieved were whole new horizons in interaction and editing! Complex request lines, having possible errors, were no longer among the user's tools. As the effect of commands became visible as each command was typed, erroneous action could be stopped at the erroneous command. This natural, simple interface was eventually the one adopted (independently) by "stand-alone word processing machines" (Wang, Xerox, Lanier, etc.). Within some time, it became apparent that Real-time edit mode was in fact a more potent and natural approach to text editing than the conventional interactive (even video-oriented) text editor.

The end result of Control R mode is the user operating as though he or she were "editing the screen" by typing keys. As a key is typed, the text on the screen changes. There are Control R mode commands to position the cursor to different character positions (and therefore lines) in the text buffer: if an attempt is made to position to some line not on the screen, TECO chooses some new portion of the buffer to display automatically. Unlike "editing terminals," the user is not *in fact* editing the screen, and thus need never be concerned with what particular portion of the buffer is in fact on the screen. The user need never "read in a new screen from the computer," "send the screen to the computer," or any similar implementation-derived constraint.

Control R mode depends upon the ability to interact on a character-at-a-time basis with the editor program (TECO). As each single character is typed, TECO must take action, both modifying the buffer (the text being edited) and updating the display. This interaction is innately more "expensive" (consumptive of computer resources) than the line-at-a-time request lines of conventional editors. Much of the historical interest in the development of Multics Emacs derives from the necessity for this character-at-a-time interaction and ways of ameliorating its performance impact.

The next significant advance was the introduction of the ability to associate arbitrary *macros* (TECO programs, possibly user-supplied) with keys in Control R mode. This ability allows arbitrarily complex actions to be performed in response to any given keystroke. TECO excels at the ability to construct subroutines, of editor commands, in its internal registers. These subroutines can call each other, recursively, and pass arguments and results around. It is quite common in TECO to construct these macros to perform arbitrarily complex text-manipulation tasks, such as dealing with sentences, paragraphs, or constructs in the source representations of specific programming languages. The ability to associate TECO macros with keystrokes allowed editor developers and users to augment Control R mode by adding sets of keys whose functions are tuned to specific editing tasks.

By 1976, several packages of TECO macros [TMACS, TECMAC] had proliferated. These packages contained macros for many common operations on text entities, and brought the power of Control R mode up to and beyond that of today's stand-alone word processors. By this time, use of "raw TECO" had almost ceased, with almost universal use of these Control R mode macro packages. TECO augmented by these packages in fact transformed the user's terminal into a word-processing, program-processing, mail-processing, or other highly specialized video device.

At this time Richard Stallman coalesced most of the ideas in these packages, and created a unified approach to command set, extensibility, documentation, and integration of these facilities, and created a large, unified, set of macros which came to be known as Emacs. The name is derived from "**E**ditor **Macros**."

Since the user using Emacs (or any of the earlier packages) never deals with the command set of TECO, but only the actions specified by the Control R mode macros, Stallman rightly considers Emacs to be an editor implemented in TECO, the latter being an interpretive editor implementation language, and one suffering severe deficiencies at that. In fact, Stallman's viewpoint is largely justified, and one that we will adopt. The impact of Emacs and Emacs-style editing far outweighs that of any TECO, and the basic philosophies of TECO, as a user interface, are largely masked by Emacs. Indeed, TECO is simply the vehicle in which ITS Emacs is expressed.

It was soon found that Emacs could be taught within minutes or an hour to those with no technical experience at all. Experienced and sophisticated users found Emacs to be eminently more usable than any of the previous forms of editing, and via the construction of more macros (for processing Lisp source programs, for example), could rapidly be extended to handle any task in the same manner. Emacs rapidly became the standard editor on ITS, and has remained so to this writing.

# III. Inception of Multics Emacs

When Multics was begun in the mid-1960's, the legacy of CTSS left a large variety of IBM half-duplex printing terminals as the standard interactive device. No display terminals were used on Multics (other than storage-tube graphics devices (which cannot be used for video editing)) until the mid-1970's, when the consumer display terminal first became readily available. Some Multics users purchased these terminals (notably the Delta Data 4000, which was the *only* kind of terminal in use at the Multics installation at the University of Southwestern Louisiana), using them either as "glass teletypes" or via "local editing." Video terminals generate less noise and less waste paper than printing terminals, and, for higher line speeds, are markedly less expensive. Users of video terminals on Multics developed methodologies of using qedx or Falksen's editor to print lines to the screen, and go into "input mode,"

whereupon they would edit the screen by pressing locally-provided terminal controls, and then depress the "transmit" button, sending screen contents over. So prevalent were these techniques that some users of Multics Lisp maintained no source files, using local editing to edit functions and read them back in. Such poor programming practice, and the severe limitations of the "command set" of the "built-in editors" of terminals created poor results all around.

Eugene Ciccarelli, at MIT's Laboratory for Computer Science (LCS) (now at BB&N), having been a major contributor to the pre-Emacs TECO macro packages, and then to ITS Emacs, felt the need for character-oriented interaction on Multics, and in 197?, developed a technique which relied upon the Multics ARPANET [ARPANET] implementation to effect such input. The Multics ARPANET implementation was not limited to line-at-a-time I/O. Ciccarelli constructed a video-terminal oriented line-editor, using Control R mode-like commands, on Multics. A user of his system would log into Multics via the ARPANET, and then invoke Ciccarelli's line editor. From this editor, the user would log into Multics recursively, via a looped-back ARPANET connection. Ciccarelli supported three popular terminals. His technique allowed users to edit any Multics input line, and retrieve previous lines. It acquired some limited use at LCS, but did not become popular, the use of two processes and the ARPANET being neither readily available to most users, nor in any way efficient. Some experimentation was done by Ciccarelli and Charles R. Davis with developing a video editor in this environment, but it never saw any use or left the experimental stage. Ciccarelli's efforts were the first use of character-at-a-time input on any form on Multics.

By late 1977, people at MIT, familiar with the video editing developments on ITS, began to see the state of editing on Multics as severely deficient. Although various proposals to create new qedx-like editors were being bandied about, none of them achieved sufficient consensus to be implemented. The people critical of editing on Multics were not impressed by any of these proposals, and realized, from experience on ITS, that the real-time video editor (as ITS Emacs was now recognized to be) was rapidly becoming the central user interface of the system which supported it. They began to claim that Emacs symbolized everything that was right about ITS and wrong about Multics.

The numerous and vocal partisans of Multics, which has traditionally prided itself on its user interface, were sensitive to this criticism, and a discussion rapidly grew about whether or not Multics could ever support an Emacs-like editor. (An *Emacs-like editor*, for the purposes of this discussion, is a real-time, character-at-a-time, interactive, display-oriented editor (control R mode-like) running in a mainframe). A large set of people contended that this was the wrong way to go, given the predicted economies of terminals, processors, and communications equipment. These parties stated terminals would become more intelligent, and would be more adept at the editing task as technology advanced. Others, notably Stallman, contended (correctly in the author's view) that only by an integrated, powerful program running in a mainframe, could an interface of sufficient usability, generality, extensibility, and power be supported.

Other discussion centered around the feasibility of character-at-a-time input on Multics. Multics had never supported such: neither do many large commercial mainframes. DEC, from their minicomputer orientation, provided this facility, and implementors of operating systems on DEC hardware seem predisposed to carry this facility through. The Multics Communications System is complicated by the presence of a Front-End Processor (known as the *FNP*), which performs line-control and echoing functions. The mainframe/FNP communication protocols were not optimized for small transfers, or rapid interaction, and FNP interrupts to the Multics central system are expensive. Process wakeups on Multics are expensive as well, and the working sets of Multics processes are large. The predicted

expense of character-at-a-time interaction was great, yet the fundamental importance of this feature to the type of system under contemplation was acknowledged by all.

By February, 1978, the discussion had basically come down to whether or not someone in the Multics organization could be persuaded to attempt to implement such an editor. The author had used ITS Emacs during the preparation of a course that January, and had become convinced of the unquestionable superiority and desirability of the ideas contained therein. A demonstration of ITS Emacs, using the Delta Data 4000 at Honeywell's Cambridge Information Systems Lab, was scheduled for March 3, 1978, with the intent of stimulating interest within the Multics Development organization in real-time editing.

The demonstration was widely attended. However, lack of preparation, hardware difficulties, and the presence of a number of unruly outsiders led to chaos, and not very many people went away with any ideas markedly different from that which they brought to the demonstration. One person who was impressed was Larry Johnson, a Multics Communications specialist, who was sufficiently impressed that within in hour after the end of the demonstration, he had effected a patch to the FNP to cause transmission of each character to the mainframe as it was typed. This implementation was extremely inefficent, causing interrupts and wakeups on every character typed, yet it inspired the author and the ITS people present to consider character-at-a-time I/O as no longer a possibility, but a reality. If an implementation of a real-time editing system could be constructed, it was felt that it would rapidly gain such popularity that all necessary implementation efficiencies would be necessitated by popular demand.

It was that day that the author contemplated the construction of Multics Emacs. Several lessons were to be learned from the history of ITS Emacs. The inferior nature of TECO as an implementation language, plus the "add-on" nature of the display support, led to consideration of an editor constructed with real-time display editing as its primary goal. The removal of the overhead of interpretation of an intermediate code (e.g., TECO) seemed to be one way in which efficiency could rapidly be bought back. A goal-directed implementation in PL/I, the standard Multics system and application language, suggested itself. Multics PL/I is extremely complete, well-debugged, and optimized to systems programming tasks [AG94].

However, another of the lessons learned from ITS Emacs was that of extensibility and modularity. The ability to add programs (macros, in the TECO implementation) to augment the editor was crucial. If the smaller primitives of the editor were to be called in an efficient fashion, they would have to be internal subroutines (internal procedures) of some very large program, which would presumably grow without bound as functionality was increasing the difficulty of maintenance, and incremental growth and debugging. Calling external programs is expensive on Multics. If this approach were taken, the ability to extend function by writing new code would require externally available interfaces to the editor primitives, which would presumably operate at reduced efficiency, never equalling the efficiency of code in the "big" procedure. This appeared to be an unreasonable restriction. Other alternatives which presented themselves, such as lexical inclusion of the code for editor primitives, seemed equally unpalatable.

The incremental overhead of creating a new PL/I source program in a large subsystem is substantial: a large amount of communications information must be lexically included in the source program for even the smallest component. Declarations are necessary for every temporary variable, every built-in function, and every other procedure name used. The substantial expense of the PL/I compiler must be invoked on every version of code, no matter how minor the change, to test it. Either source programs

proliferate without bound, or routines must be packaged into larger programs, increasing the compilation overhead without bound. Any possible user augmentation, a prime feature, would have to include a large amount of declarative information giving internal structure of editor data-bases.

PL/I did not seem conducive to interactive development of such a subsystem. The choice of languages was limited: technology had long since passed the stage where assembler language was reasonable, and the idea of substantially augmenting Multics TECO seemed distasteful, and a step backwards. Among the other standard languages on Multics, neither FORTRAN, COBOL, APL, nor BASIC could be given serious consideration.

It was at this point that the idea of Lisp as an editor implementation language occured to the author. Lisp function calls are substantially more efficient than Multics PL/I external calls (but far less efficient than PL/I internal calls). Lisp is ideally suited to buiding large subsystems of many functions, many of them very small. Lisp, when viewed properly [Lispbook], is a highly object-oriented language, masterfully adept at maintaining complex data structure. Programs can be added to a running Lisp environment without the need for any compilation at all (when debugging), and can be interactively debugged with the aid of the Lisp interpreter. The global namespace of Lisp allows users to reference only the variables they need to reference for a given task, without compiled-in assumptions as to their storage location in any structure. Lisp seemed to solve many of the design problems inherent with the use of PL/I.

The author also had substantial experience with the Multics implementation of MacLisp [Moonual], having implemented a large subsystem (a Multics crash-dump analyzer) in it, and becoming sufficiently dependent upon it that he had, by this time, become the maintainer of the implementation.

Lisp has acquired a bad reputation for several reasons, all false. People often point out the inefficiency of interpreted code. However, all production Lisp code in mature implementations is compiled into machine code, not any kind of pseudocode. The Multics Lisp Compiler [LCP] is well-debugged, and reasonably efficient. People speak of the innate inefficiency of the data-representation of Lisp: yet, the existence of character strings as a data type (in MacLisp), plus the judicious use of outside-of-the-language facilities where appropriate, create a reasonable efficiency of data representation. People are quick to point out the unreadability and unmaintainability of the source code of Lisp programs; yet, it is precisely through editors like Emacs that automatic technologies for editing and formatting Lisp code come about. Given proper editing tools and formatted code, Lisp is more readable (in some opinions) than many block-structured languages. Finally, many, if not most, Lisp implementations, are not fully mature, and thus not suited for large subsystem implementation. The unique nature of the Multics process environment allows subsystems in Multics MacLisp to invoke, and communicate readily with, Multics facilities outside of the Lisp environment. The set of facilities available to a program running in a Multics process is one of the major features of Multics.

The status of Multics MacLisp at this time was that of a holdover from MIT co-development days of Multics. Other than the above-mentioned dump analysis tool, it had no major uses, and its documentation [Moonual] was no longer published or available. Honeywell did not support it officially in any way. No distributed Multics program was written in Lisp.

The decision to use Multics MacLisp as an implementation language had profound consequences for its future. A large amount of interest in Lisp was eventually manifested by those desirous of knowing it solely to be able to augment Multics Emacs. Non-technical personnel have been observed to acquire enough knowledge of Lisp to extend Emacs for only this purpose. Lisp had previously been limited to

undergraduate computer science courses and Artificial Intelligence laboratories. The choice of Lisp turned out to be a very wise choice, for the incremental creation of the editor, through its "extensibility," could have been done in no other language.

On the evening of March 3, 1978, Bruce Edwards and the author sat logged into MIT's Multics System and coded, and debugged, what was to become the central module of Multics Emacs. This Lisp program maintained a buffer of text as a list of **Editorlines**, each **Editorline** representing one line of text of the buffer. An **Editorline** is a triplet, of a **Linecontent**, the previous **Editorline** of the buffer, and the next **Editorline**. A Lisp variable identified the *current* **Editorline**, and another variable the *current point* in the current line. The **Linecontent**s were, in this initial implementation, represented as a doubly-linked list of single characters. The current `point` in the current line was identified with the list node containing the character at that point Lisp functions were provided to insert a character at the current point, delete a character, break a line into two by inserting a newline, merge two lines by deleting one, move the "current line" and "current character" pointers forward and backward over characters and lines, and print out a line.

# IV. Multics Emacs: The Embryonic Phase and Basic Decisions

The program created that day did nothing but maintain a buffer of text in Lisp list structure. Primitives to operate upon the buffer and the pointers had no visible effect when invoked. Lines had to be "printed" by invoking a function provided for that purpose. This program is (much augmented since) the core of Multics Emacs. It could have been used equally well to implement a non-display editor of the conventional (qedx, or TECO-like) mold.

The basic structure of an Emacs-like display editor consists of three parts, such a program (called the *basic editor*), an *interactive driver*, and a *redisplay*. The basic editor maintains text and contains primitives to operate upon it. The interactive driver interprets characters read from the keyboard, and invokes primitives in the basic editor to execute these requests. The redisplay is a screen manager, and is invoked by the interactive driver between input characters, and constructs and maintains a display screen by looking at the basic editor's buffer, and effecting incremental change. Much more will be said about the interactive driver and the redisplay later.

The basic editor developed on March 3, 1978 had several noteworthy design points. The representation of the text buffer as a list of **Editorline**s was designed to optimize the redisplay of a display editor, and to optimize user-visible and internal primitives which dealt with lines. Much of editing and text-processing operation consits of operations upon lines, or iterated over lines. Having to search for end-of-line characters for these common operations seemed suboptimal. More specifically, the redisplay would have to identify lines of the buffer with lines displayed upon the screen when the screen was last updated, in order to move lines around and better know which text to compare at redisplay time. Redisplay is basically a compare-and-update process (which will be discussed further), and any way of making the comparisons and heuristics cheaper is of tremendous value.

The maintenance of the buffer as list-structure also means that text does not have to be moved around to perform insertions or deletions: lines deleted are simply unlinked from the list representing the buffer, and new lines are linked in. Lisp garbage collection ultimately reclaims the space used by deleted nodes. Yet, even in non-garbage-collected programming languages, explicit storage management of the list nodes allows this potent strategy to be utilized. The doubly-linked list has as a disadvantage that the

representation of an empty buffer, i.e., one containing no lines, and no characters at all, is difficult, and this remains a problem to this day. Multics Emacs buffers are created with one line consisting of only a newline character; such a buffer is considered to be "empty," and the reading of a file into an empty buffer is special-cased to produce the desired result. Another problem with this approach is the difficulty of searching for a character string, particularly one containing imbedded newline characters. In spite of these difficulties, Daniel Weinreb subsequently adopted the doubly-linked buffer list in his editor ZWEI [DLWThesis] on MIT's Lisp Machines [Chineual]. ZWEI is also coded in Lisp.

The decision was made from the start to represent **Linecontent**s in two different ways, one way for all lines but the current line, and the other for the current line. The representation medium for the current line must be easily modifiable, while the other lines must be storage-efficient. MacLisp strings, which were the natural choice to represent **Linecontent**s, are not modifiable. (Strings on the Lisp Machine [Chineual], on the other hand, are array-like objects, and are). In the initial implementation, the current line was represented by the doubly-linked list of characters described above. The **Linecontent**s of all other lines were (and still are) Lisp strings. The current line is copied into the modifiable representation when it is first modified (this is known as *opening the line*), and copied back into a (new) Lisp string when the current line is left (i.e., is no longer the current line). This strategy matches fairly well the normal user pattern of moving to a line, modifiying it, moving to another line, etc. Making sequential changes through a buffer, or simply typing in any amount of new text, are both special cases of this pattern.

It was realized quite early that the doubly-threaded list of characters could not be efficient enough for a production implementation: even attempting to re-use the storage of the list nodes of the representation was deemed too inefficient. For operations upon the current line, the traditional character-string buffer had distinct advantages. Eventually (about three weeks into the development of the editor), a new type of Lisp object had to be invented to hold the modifiable representation of the current line. The *rplacable string* (from the Lisp terms *rplaca* and *rplacd*, the primitive pointer storage modifiers) is stored outside of the Lisp environment, in a Multics segment. Pointers to it can exist in Lisp pointer cells, and these pointers have the type bits of a character string (Multics Lisp pointers are explicitly typed). A special bit pattern in the pointer indicates that the pointer must not be chased or be subjected to data-object relocation by the MacLisp garbage collector, which, in the Multics implementation, is a recompacting-type garbage collector [GCPAPER]. Four rplacable strings are needed by the entire implementation.

The rplacable string is manipulated by two kinds of primitives: normal Lisp (and Lisp-interfacing) primitives can view it as a character string, and special primitives (in LAP, the Lisp-interfacing assembler in many implementations) are provided to modify its contents: delete characters from any point in it, and insert characters at any point in it. These primitives make use of powerful Multics hardware instructions which can perform overlapping string moves in either direction, mapping precisely the actions of inserting and deleting characters from the active current line. The LAP functions run in the Lisp environment, and are called as efficiently as one Lisp function from another.

The next step was the development of an interactive driver. The function of the interactive driver of an Emacs-like editor is basically that of TECO Control R mode: to read characters from the user's keyboard, find out what program (in TECO, a macro, in Lisp, a function) to run (the *binding* of the key), and execute it. After each such function is executed, the interactive driver invokes the redisplay to reflect changes to the buffer on the screen. This loop of read a character, dispatch on it to a function, redisplay, repeated indefinitely, is the basic control loop of an Emacs-like editor.

The interactive driver provided no special problems in the initial implementation: the atomic symbol [Moonual] whose name was the character which had been typed was given as a Lisp property the function to be run when that character was struck. Two-keystroke sequences were mapped into different properties of the second character. This mechanism was not conducive to switching key bindings rapidly nor easily, and prevented the latter from being implemented for the three months while it lasted. The storage inefficiency implicit in the storage of the properties was also undesirable. Nevertheless, the natural mapping of the key-bindings into the Lisp property mechanims provided an easy path to create an operative mechanism to allow the rest of the editor to grow.

A more significant difficulty was the availability of character-at-a-time I/O for experimentation. The growing editor was completely experimental and not part of any recognized or funded project, and no resources were immediately available among the already highly overcommitted Multics Communications support specialists. Thus, Ciccarelli's code was sought out, to find the basis of his techniques of single-character-input via the Multics ARPANET. Within a day, the interactive driver was operating in true character-at-a-time (real-time) mode for processes logged in via the Multics ARPANET. As there was no redisplay, a Lisp function which printed out the current line, with an overstruck < and > for the current character position, served in its place.

Soon thereafter, Johnson's patch to effect single-character transmission from the FNP was applied on a regular basis, on the CISL Development Multics Site (a testbed system with no real user community) as editor development progressed. As this went on, users on MIT's Multics system willing to experiment with the new editor were forced to use it via the ARPANET, for those administratively allowed to use the ARPANET, or experiment with it in non-real-time mode (typing a linefeed to activate, i.e., cause Multics to take cognizance of) previously typed input. These development paths proceeded in parallel: the application of the patches (to become known as the *breakall mode patches*), a slow and dangerous operation requiring highly privileged access, to the MIT service system was out of the question.

The next and final step in the birth of the editor was the design and creation of the Redisplay. A redisplay's job is to depict a section of the contents of a buffer on a video terminal screen. The redisplay is invoked by the interactive driver after each function has been called, which has performed arbitrary modifications upon the buffer. The redisplay must know precisely what it put upon the screen the previous time it was invoked, compute what must be put upon the screen this time, and determine how to most effectively and efficiently modify the screen, using the terminal facilities available, to transform the old screen content into the new screen content.

It is one of the fundamental design principals of a real-time video editor of this nature, that the basic editor is aware of neither the existence nor the organization of the redisplay. The redisplay is, symmetrically, aware of neither the organization nor actions of the basic editor. Between invocations of the redisplay, it is given no hint as to how the transformations upon the buffer which it will observe were performed, it can observe only the new state of the buffer (including current line pointer, etc.). This philosophy leads the highly desirable state of affairs, where *extensions* (i.e., user-supplied editor features) as well as the basic editor need not be at all concerned with display management, but only manipulation of the text in the buffer via the supplied primitives. The display is managed automatically.

The first coding of the Multics Emacs redisplay was performed on March 6, 1978. The Delta Data 4000 terminal at CISL was the only video terminal readily available, and fortunately, in spite of severe implementation bugs in the terminal, it had the features of the better class of consumer video terminals available. The ability to insert and delete characters and lines from the screen was thus designed into the

redisplay from the start. The interface to the redisplay was designed as one Lisp function, `redisplay`, which took no explicit arguments. The current buffer, the current line pointer, the current character position pointer, etc., i.e., the current state of the basic editor, are all implicit parameters. The *contract* of the function `redisplay` is to determine what is to be displayed on the screen, how it differs from what is already on the screen, update the screen, and remember for next time what is *now* on the screen.

Even the appearance of simple typed input on the screen is a manifestation of the redisplay. Theoretically, a redisplay occurs between every two characters typed, and it is the redisplay which puts all characters on the screen, including typed input, one at a time, as they are typed. In fact, later optimizations (to be discussed) allow participation of the operating system to be negotiated for simple echoing, but the principle remains the same.

The redisplay is the only part of the editor which interfaces to the terminal's display. In such a system, the keyboard and display of the terminal are considered to be completely disjoint; terminals for which this cannot be said to be so are simply not usable in this environment. Device-independent terminal support was provided by supplying a separate Lisp program (known as a *CTL*, for "controller") for each terminal type supported. The functions defined in Each CTL are the same, and provide the common functionalities of terminal displays. For example, `DCTL-position-cursor`, of two arguments, coordinates, positions the terminal's cursor to those coordinates. `DCTL-delete-lines`, of one argument, the number of lines to be deleted, deletes that many lines at the current cursor. An initialization function, `DCTL-init`, is provided in each terminal's CTL, to set parameters used by the redisplay which state which functions are available: all terminals subset the maximal CTL. Functions in all CTLs call a common interface to output characters to the terminal.

Via the CTL mechanism, the dynamic nature of the Lisp workspace, and of Lisp function calling in particular, is used to add a terminal-specific component to the editor at invocation time. The type of terminal being used is (usually) provided by the Multics Communications software, and thus, the loading of the CTL is automatic.

The heart of the redisplay is its *screen image*, a data structure by which the redisplay remembers what it left on the screen after one invocation, so that it might know what is there at the start of the next. Rather than an array of characters, the representation of the screen image is designed to take advantage of the basic editor's division of the buffer into **Editorline**s. The screen image is a Lisp array [Moonual] of one dimension, one with element for each physical line of the display terminal. The element of this array (called `screen`) is a **Displayline**, which is a pair of an **Editorline** and a string which is the exact character sequence known to be on the physical line of the display (the **Linedata**). Several consecutive **Displayline**s may have the same **Editorline**, for **Editorline**s whose printed representation is longer than one physical terminal line.

The contents of a **Linedata** reflect the actual printed representation of an **Editorline** or a part of an **Editorline**. **Linedata**s do not contain new-line characters, tabs, non-printing characters, or other format effectors. The number of characters in a **Linedata** is its width in columns upon the screen; all characters in it are single-width printing characters. The character in a given position on the corresponding terminal line is the character in the corresponding position of the **Linedata.** Non-printing characters in the **Editorline** (actually the **Linecontent** of the **Editorline**) are converted to their printable representation (an ASCII WRU to either "\005" or "^E" (control E), according to the Multics or ITS conventions, a user option), and tabs are converted to the appropriate number of spaces.

Note that this strategy is not readily extensible to representation of overstrikes or multi-width fonts: although Multics does not now have multi-font support (there are certainly no multi-font terminals readily available, from the viewpoint of Editor use of multiple fonts), but the lack of representability of overstrikes has been a problem (underlined text is quite common in Multics). Perhaps an addition of an overstrike map of some form to the **Displayline** structure is the solution.

The basic flow of the redisplay is to compute, for the whole screen, what the **Displayline**s of the new screen will be, filling another array (`newscreen`), update the screen based upon comparing the arrays `screen` and `newscreen` by calling functions in the CTL, and copy the array `newscreen` into screen. In fact, this operation is performed on a per-window basis every time redisplay in invoked. (A window is a section of the screen assigned to the display of a particular buffer: although there may (today) be any number of windows, the limitations of conventional terminals restrict them to be as wide as the screen.)

The redisplay contains many optimizations: it is that part of the Editor where effort in optimization is most well-spent. Optimization of line-transmission time, i.e., the least screen-rewriting for each redisplay, is most visible to the user. However, the frequency of invocation of the redisplay, and the conceptual complexity of its task, make the computation time of the redisplay a prime target for optimization, a of critical significance of the entire editor. CPU time consumption mainfests itself as system loading, billable user expense, and reduced response time visible to the user. In the Lisp implementation, where strings and list nodes are dynamically allocated, minimization of storage generation for each redisplay is an equally important consideration. The detailed algorithms and optimizations of the redisplay will be given in Appendix A.

With the completion of the initial implementation of the redisplay, the basic loop of interactive driver-basic editor-redisplay was functional, providing an operative editor. The next necessary addition was that of *extensions*, to provide useful function. An editor *extension* is a body of code which provides text-manipulating capability specific to some domain of text processing, and operates by invoking primitives in other extensions and the basic editor. An extension does not know about the data formats or organization of the basic editor. Extensions are completely unaware of the existence of the display and the redisplay. Like the basic editor, extensions manipulate text in buffers (by calling the basic editor, however, as opposed to actually manipulating the editor data structure), and the display "follows automatically" without any explicit coding thereto.

Extensions are often written by users, although a set of "extension code" is incorporated into the editor (the primitives for manipulating words, sentences, paragraphs, comments, and whitespace). Further supplied extensions, which are *loaded* into the editor environment on demand for the Emacs library, include code knowledgeable about PL/I or Lisp syntax, the Emacs mail system, and other optional packages. Extensions are written in MacLisp, augmented by a set of Lisp Macros [Moonual] which tune the syntax of MacLisp for data and control constructs necessary for manipulation of the Multics Emacs environment. The extension language is an interesting one, and will be covered in more detail in Appendix B.

It is considered to be a major feature that the language of extensions is a fluid and powerful language, specifically designed for writing programs. This contrasts dramatically with the use of TECO for extension coding in ITS Emacs. While ITS TECO code is quite baroque, and accessible to only a few experts, the extension language of Multics Emacs is sufficiently natural and simple to learn that non-technical personnel have successfully written and debugged non-trivial extensions. Stallman [Stallman] concurs about the relative merits of TECO versus Lisp as an editor implementation/extension

language.

Multics Emacs provides a number of features in its Lisp-program-editing mode to facilitate the interactive development and debugging of extensions as they are being written: functions in an editor buffer can be added to the editor environment in a keystroke. Most of the extensions after the first few weeks of Multics Emacs have been developed in this way. The interactive, incremental development of Lisp programs in this way is not unlike the techniques developed on MIT's Lisp Machine [Chineual, DLWThesis].

Among the more interesting of the initial features provided for the use of extensions was the creation of *character sets* as extended-type objects for extensions trying to utilize the character-scanning capabilities of the Multics hardware. A primitive in the basic editor returns, in exchange for a character string, a pair of two Lisp arrays designated as a *character set*. Another set of primitives in the basic editor, given a character set, will scan the buffer from the current point, backwards or forwards, possibly only in the current line, for the first character *in* the character set, or *not* in the character set, as required. The arrays which constitute the character set are set up for immediate use by powerful Multics hardware instructions (of the "translate and test" variety) that actually are used to perform the scanning.

The ability to deal with external files was also provided naturally by Lisp type-extension (the ability to arbitrarily define a type of object created by composition out of other, possibly primitive object types). **Editorline**s whose contents were unmodified strings in external files have **Linecontent**s which are pairs of a pointer and length (as a list node), the pointer being to the location of the character string (the Multics hardware supports character addresses in pointers) in the Multics Virtual Memory. The lowest-level LAP primitives will deal with either a Lisp string or such an object (a *Filecons*) identically, as being representations of character strings. Thus, the read-in of a file to Multics Emacs does not cause storage generation for copying of all of the strings in the file into MacLisp strings.

A facility needed by most extensions is that of a *mark*, or buffer pointer [DLWThesis]. A mark is an identification of a given character position in a given buffer. Extensions use them liberally, and at least one mark is a user-visible feature. Marks are used to identify points in the buffer, and pairs of marks, or the current point in the buffer and a mark, are used to identify regions of the buffer. Marks are problematic as they need to be relocated as text surrounding the points they represent is modified or deleted. This is to say that a mark which identifies some character on some line must be made to identify the nearest remeaining character or line if that character (more specifically, one before it on the line), or the line itself is deleted. This imposes an overhead on almost every buffer-modification operation, as a check must be made to see if any marks in the current buffer need be relocated on account of it.

Multics Emacs' approach to the minimization of this overhead is to associate marks with lines by representing a mark as a pair of an **Editorline** and a position within that line. A mark is created by a primitive in the basic editor which creates a mark designating the now-current editor `point` in the buffer, and both returns it and keeps it in a list of marks associated with the current buffer. When a line is opened for modification, the list of marks for the current buffer is scanned for marks designating the current line, and a secondary list of marks (marks for the current line) is developed. It is only this list which is scanned when the buffer is modified, for only the current line is ever capable of being modified. Marks are expected to be "freed" (i.e., removed from the per-buffer list and the secondary list) by the code that created them when they are no longer needed---this keeps the overhead of marks from growing limitlessly. Due to the definition of buffer position in Emacs (between characters), the simple and tremendously important case of adding text at the end of the current line can never cause the

relocation of any marks, so the secondary mark list need not even be scanned for this most common modification.

An interesting alternative strategy for marks is implemented by Weinreb [DLWThesis] in his ZWEI editor: each Editorline of his data structure includes a list of marks for that line. This avoids having a per-buffer mark list or ever having to scan it. The per-**Editorline** overhead implied in his approach, however, was deemed too great for Multics Emacs (list nodes are much more expensive in Multics MacLisp (144 bits) than the Lisp machine (32 bits)). Weinreb also admits "non-relocatable marks," i.e., marks not put on any mark list, which are valid only during a time during which the invoking code is under obligation to cause no buffer modifications.

Once the core of the editor and supplied extensions were operative, development proceeded along three separate paths simultaneously: augmentation of functionality, performance improvement, and new interactions with the ARPANET. These areas will all be discussed independently.

# V. Early Enhancements

Within the first two months, the need arose to support many text buffers simultaneously within the editor. People usually edit many files at once, and the conventional Multics editors provided this capability. In addition, many specialized uses of buffers were to develop in time, supporting many exotic features of the Editor.

The implementation of multiple buffers was viewed as a task of multiplexing the extant function of the editor over several buffers. The buffer being edited is defined by about two dozen Lisp variables of the basic editor, identifying the current **Editorline**, its current (open/closed) state, the first and last **Editorline**s of the buffer, the list of marks, and so forth. Switching buffers (i.e., switching the attention of the editor, as the user sees it) need consist only of switching the values of all of these variables. Neither the interactive driver nor the redisplay need be cognizant of the existence of multiple buffers; the redisplay will simply find that a different "current **Editorline**" exists if buffers are switched between calls to it. What is more, the only functions in the basic editor that have to be aware of the existence of multiple buffers are those that deal with many buffers, switch them, etc. All other code simply references the buffer state variables, and operates upon the current buffer.

The function in the basic editor which implements the command that switches buffers does so by saving up the values of all of the relevant Lisp variables, that define the buffer, and placing a saved image (a list of their values) as a property of the Lisp symbol whose name is the current buffer's. The similarly saved list of the target buffer's is retrieved, and the contained values of the buffer state variables instated. A new buffer is created simply by replacing the "instatement" step with initialization of the state variables to default values for an empty buffer. Buffer destruction is accomplished simply by removing the saved state embedded as a property: all pointers to the buffer will vanish thereby, and the MacLisp garbage collector will take care of the rest.

The alternate approach to multiple buffers would have been to have the buffer state variables referenced indirectly through some pointer which is simply replaced to change buffers. This approach, in spite of not being feasible in Lisp, is less desirable than the current approach, for it distributes cost at variable reference time, not buffer-switching time, and the former is much more common.

One of the most interesting per-buffer state variables is itself a list of arbitrary variables placed there by

extension code. Extension code can *register* variables by a call to an appropriate primitive in the basic editor. The values of all such variables registered in a given buffer will be saved and restored when that buffer is exited and re-entered. The ability of Lisp to treat a variable as a run-time object facilitates this. Variables can thus be given "per-buffer" dynamic scope on demand, allowing extensions to operate in many buffers simultaneously using the same code and the same variables, in an analogous fashion to the way Multics programs can be executing in many processes simultaneously.

Emacs (both Multics and ITS) supports the notion of *modes*, which are specific tunings of the editor's interface for specific tasks. These tunings include sets of key bindings, and settings of the user-visible per-buffer variables. For instance, in PL/I mode, the "Tab" key/character means "Indent the current line correctly according to standard PL/I indentation," and Column 61 is the column where comments are placed. In Lisp mode, "Tab" means "Indent the current line correctly according to standard Lisp indentation," and comments go in column 51. Modes take effect on a per-buffer basis: each buffer is in a given mode, *Fundamental mode* being the initial default. Modes are one of the most significant advances of Emacs over other similar editors.

The key to the ability to support per-buffer modes is the ability to change key bindings rapidly and conveniently when buffers are switched. Implementing this meant replacing the key-binding mechanism that used Lisp properties and replacing it with a Lisp array to dispatch the interactive driver. The basic structure used is a Lisp array with 128 elements, one for each possible ASCII character. The array elements can contain either the Lisp constant `nil` (indicating that the key is "undefined"), a Lisp symbol which defines a function to be executed for that character, or a pointer to another similar array of 128 elements, for characters which are *prefix characters* (non-terminal characters of multi-character sequences). For historical reasons, the prefix character "ESC" (ASCII "escape") is special-cased, having its dispatch-vector being a second row of the root-node dispatch vector.

A list is kept, per-buffer, of keys whose bindings were changed while in that buffer. The element of the list gives the key (as a path through the dispatch vectors) and a binding. A key's binding is changed by placing it in this list, placing the original binding from the dispatch vector in the list element, and changing the element of the appropriate dispatch vector. When the buffer is exited, the list element and dispatch vector element are swapped, for each element of the local key-binding list. When a buffer is entered, the same swapping is performed, reinstating local key bindings of that buffer. A command is available to set a key in the dispatch vector without placing it in the local list: this constitutes setting the key globally (for all buffers where not explicitly defined otherwise).

This approach was designed to have the following desirable characteristics: it is optimized to buffers that have some, a few, but not most key bindings different than the default. Thus, the overhead of per-buffer dispatch vectors does not exist, plus the concomitant problems of making global changes to them. The overhead of switching into or out of a buffer is proportional to the number of key-bindings different from the default. Changes to the global default are made trivially. Clearly, the disadvantages of this approach are the overhead involved in buffers where almost every key is different (for example, all the normally "trivial" keys, the normal printing characters), and fairly odd machinations and manifestations,as well as definitional issues, when an attempt is made to change the global definition of a key which has been redefined locally)

A major feature of modern video systems is that of dividing up the screen into *windows*, or regions in which different activities are being displayed. Highly advanced video systems [Chineual] [PARC] often have dozens of windows on the screen at once, some only partially visible. In an editor, multiple

windows allow several documents to be edited while being viewed simultaneously (such as new and old versions, etc.). In Emacs, mail-reading and responding can be going on in two windows, or buffers containing interactive messages from one or more other users can be on display while other activities are proceeding in parallel on the screen. ITS Emacs supports up to two windows on the screen, so there was competitive incentive to support multiple windows in Multics Emacs.

There are two user-level window-management schemes available in Multics Emacs, the default (*static windows*), and an experimental one (*pop-up windows*), modelled loosely after the display software at Xerox PARC [PARC]. As the relative merits of the two schemes are not yet clear, use of the experimental scheme is a user option.

In the static scheme, the user creates and destroys windows by explicit command. Window sizes are set by explicit user command (a special subsystem, the *window editor* assists in this operation). All activity, including switching to new buffers, occurs in one, *selected* window, until the user selects another window, explicitly. However, certain commands take advantage of multiple windows, by attempting to place buffers on display in other windows besides the selected one. For instance, the *reply* command, issued while reading mail, builds the reply letter in a buffer, and, if multiple windows are in use, places the reply in "some other" window and selects it (in fact, the least-recently used) unless that buffer is already on the screen in some window, in which case that window is selected. Another example of such a command is the *compile* command in PL/I and FORTRAN modes, which places compiler diagnostics in "some other" window than the source program.

In *pop-up window mode* (May 1979), all commands which switch buffers, or create new buffers, or enter any type of new activity, attempt to place a window on the screen (if there is not already a window for that buffer) somewhere, creating new windows and destroying old ones (again, on a least-recently used basis) if necesssary. If such buffer is already on the screen in some window, that window is selected. Windows are destroyed either by being replaced, explicitly by the user, or if the corresponding buffer is destroyed. Window sizes are set automatically and dynamically.

Users often find pop-up windows erratic, and unpredictable in nature, especially at low line speeds. It is still an open design issue as to whether pop-up windows are valuable in a time-sharing system accessed via communications lines, as is the degree of user visibility of the buffer/window correspondence, in which the philosophy of pop-up windows is rooted.

Multiple windows were implemented by multiplexing the function of the redisplay. Each window on the screen has on display in it one (or possible no) buffer. A buffer may be on display in no windows, one window, or more than one window (although this latter case introduces several human design and technical problems). One window at any time is considered to be the "current," or *selected* window. It is in this window that the cursor is placed by the redisplay at the end of each invocation, to indicate the current point in the current buffer. The current buffer is always on display in the current window. The interaction between buffers and windows is oft-times subtle, and small changes in the way the correspondence is managed seem to produce significant changes in the visible interface of the system.

An array gives the screen location and extent of each window, as well as the name of the buffer on display therein, and a mark of that buffer designating the *window point*, or the last "current `point`" in that buffer known at redisplay time. When the redisplay runs, it performs its compare-update operation for each window defined in the array. While the current line and current point within it are used to compute what should be displayed in the selected window, the *window point* marks are used for the

other windows. As they are marks, they are updated dynamically if need be, if the buffers in those windows are modified between redisplays.

The editor's "switch windows" command tells the redisplay to choose another specific window as the *selected window*. It *also* tells the basic editor to switch buffers, as directed by the redisplay, from the buffer name in its window array structure. The redisplay will also use the window point to determine what point in the buffer in the new window to make the current point: thus, when windows are switched, editing resumes in the new buffer in the new window at the last point it was left off, and the screen content does not change. For buffers on display in more than one window, the window point identifies the last place that was the current point for each window when it was the selected window: this allows multiple windows to be used to edit multiple parts of the same buffer.

The window management system contrasts dramatically with that in ITS TECO. In the latter, TECO at all time displays text in "the window," whose position and extent on the screen are set by Emacs (in response to user commands). Thus, by switching "the window" between, say, two alternate, non-overlapping locations, two documents may be edited at once. However, simultaneous update of windows will not occur, whether the same text is on display in more than one window, or auxiliary buffers being managed for informative display purposes by active extensions.

A thoroughly unique feature, the Emacs interrupt system, was soon mandated by the interaction of Multics Emacs with the Multics interactive console message system. Multics supports the usual two types of inter-user communication, *mail* and*messages*. Mail is "sent" by the sender to the recipient's *mailbox*, a file (segment) in the Multics storage system, and "read" by the recipient at his or her leisure. Messages, intended for communications of more transient nature, are sent using a simpler command, and are printed on the recipient's console as soon as his or her "process" (Multics control point) goes idle, which is usually when waiting for console input. (Sending mail, incidentally, sends a message of the form "You have mail," to alert the recipient that his or her mailbox ought be read.)

Messages on Multics are implemented as *process interrupts* (*event call channels* [SWG]), which cause the message-printer to be invoked by the process wait coordinator when the process goes to wait and there are messages present. When the process is awakened out of the waiting state (which will happen if a new message arrives while waiting, among other reasons), the same check will be made, and the message-printer invoked if messages are present. The message-printer normally functions by reading the message and printing out on the user's console, interspersed among the recorded input and output of the user's interactive session. In the context of Emacs, this is wholly inadequate. If such a message be printed while Emacs is being used, the contents of the screen as the Redisplay envisions them are destroyed, and the position of the cursor and the screen contents can no longer be managed effectively.

On ITS, this situation is handled by the message-printer *informing* TECO (or whatever program was in control when the message was printed) that the screen has been destroyed, and the latter must refresh the entire screen before any attempt is made to use it. This was deemed inadequate, as the widespread use of low-speed terminals would cause such liberal screen-refreshing to be very frustrating. ITS, furthermore, has an integrated screen-management system, which at least allows messages to be printed out in usable form while Emacs/TECO is in control, while Multics does not: the state of the terminal interface software while Emacs is in control, on Multics, is such that messages, when they arrived, were not printed intelligibly (this is because Emacs, in order to perform low-level terminal control, requests the terminal control software in the supervisor to suspend such control).

It was deemed best that Emacs "handle" message-printing while in control: Emacs should set itself up to be called by the message-printer when a message need be printed. The needed interface of the message-printer, to call other programs to print the actual message, is in fact provided in the Multics message system. This way, Emacs could print the message how, when and where it saw fit, integrate their printing with the screen management it performs, keep transcripts of messages received and sent, and organize response to messages in a useful way.

The Emacs interrupt system was introduced to fulfill these needs. Asynchronous events such as messages and time-driven features invoke an entry point in the Emacs subsystem which queues information about the asynchronous event, and sets a unique *interrupt cell* associated with that type of event. The interactive driver samples the interrupt cells in its basic loop. Associated with each interrupt cell is an interrupt-handler function, an extension which invokes the normal functions of the basic editor (including its redisplay interfaces) to perform the required processing. In the case of interactive messages, the handler parses the message header, places it in a unique buffer associated with the sender, and, if that buffer is not on display in a window, calls a redisplay primitive which displays it at the top of the screen in such a way that the redisplay knows precisely what portion of the screen needs be refreshed to remove it.

The occurrence of an Emacs interrupt causes the input-character reader to return: the lowest-level caller of the input-character reader has to be prepared for returns of "no characters, just an interrupt," and allow the interactive driver to sample the interrupts appropriately. This has deep implications upon the optimized echoing schemes (to be discussed) which allow partial echoing by the operating system: the operating system must be stopped from echoing when an Emacs interrupt occurs, and the known screen contents synchronized with what had been echoed up to that point by the operating system.

The Emacs interrupt system is used for console messages, a time-of-day display that updates continuously, and the receipt of ARPANET output in *User TELNET* and *User SUPDUP* modes, where a buffer and window are used to implement a remote terminal connected to a foreign computer system.

Multics Emacs has an automatic documentation system, not unlike that of ITS Emacs. Documentation on any editor command can be displayed easily via a special command which requests a keystroke to be documented. Command documentation is stored in an indexed sequential file maintained by vfile_, the Multics file manager. The keys to this index are the command names, not the keys used to invoke them. The command documentation is stored in such a format that keys are almost never mentioned by name, but rather by command name (i.e., **go-to-beginning-of-line** instead of "Control A" or "^A"). Before documentation is displayed, substitutions are made automatically for command names which are the bindings of keys in the current buffer. Thus, the documentation is accurate, no matter how the user has customized his or her key bindings. Extensions may document their commands by placing documentation as a Lisp property of the commands supplied therein: the documentation system will search there first, and substitute and display such documentation as though it had appeared in the documentation file. A special mode exists for editor maintainers to get at, unsubstituted, the documentation in the file, edit it, and add to it.

Additional documentation features, derived from ITS Emacs, include the ability to find out the last fifty keystrokes typed, and a command which matches on substrings of command names to find any commands (and the keys which invoke them) "apropos" to a given topic (for this reason, command function names must be kept reasonably mnemonic).

# VI. Echo Negotiation

As soon as Multics Emacs was introduced, people began to react negatively to the idea of a large time-sharing central processor being interrupted on every character, and waking up a Multics process on every character, just to "echo" characters (in the worst case), i.e., put typed characters back on the screen. Multics hardware price/performance is less than optimal, and working sets of Multics processes are large, and the introduction of real-time editing was viewed as a crippling blow to an already dangerous performance situation. Many observers rejected the idea of a mainframe performing the functions that they associated with "intelligent terminals:" of course, these objections are invalid, because no commercial terminal approaches even the weakest time-sharing editor in expressive power.

The simplest approach to the implementation of a real-time editor involves every single character typed being processed by the editor immediately, as has been described. This is precisely the approach taken by ITS TECO. However, ITS working sets are much smaller than Multics working sets, and ITS rarely runs more than 20 or 25 users. Character-at-a-time interaction has the most profound negative performance implications: any way of alleviating its impact is of tremendous value. The simple entry of text is the most obvious candidate for optimization. If no "errors" are made, text type-in is a tremedously cheap operation using the standard Multics editors, and until "editing" has to be done, a smooth-running interaction requiring no action by the system other than the echoing of characters. The character-at-a-time wakeup and redisplay of Emacs buys nothing in this mode (until some "error" is made) at tremendous cost.

On the other hand, the unity of the Emacs interface, and the lack of "text entry mode vs. edit mode" and similar implementation artifacts are fundamentally important features, and not to be traded off for any amount of implementation efficiency. Optimizations must be completely transparent and automatic, or they are not optimizations at all, but design tradeoffs.

During normal Multics interaction, on a full-duplex (non-local echoing) terminal, echoing is performed by the front-end processor (FNP), unconditionally. During the running of Emacs, echoing is turned off, so that the Emacs redisplay can manage the entire contents of the screen completely. The FNP cannot echo during the running of Emacs for several reasons. The redisplay cannot know accurately what is on the screen if the FNP is placing data on it as well. Nor can the FNP in general assume that echoing is valid, for type-ahead (input typed before the system has responded to all previous typing) cannot be placed on the screen until the system has responded to previous input, positioning the cursor (which determines where the next characters will be placed) as that input specifies.

FNP echoing, however, is very efficient (the FNP is a communications processor optimized for such tasks), and Emacs echoing is maximally inefficient. Thus, a scheme (*Echo negotiation*) was devised [Echnego] by which the FNP echoes conditionally during Emacs use, with protocols to address the timing problems alluded to above. When operating at maximum efficiency, the FNP echoes all characters of text being typed in to Emacs, never interrupting the mainframe or waking up the Emacs process until the end of a line or "editing" is attempted. This has the desired effect of reducing mainframe interrupts and wakeups by a factor equal to the average number of characters in a line. Implementation of this scheme involved augmentations to the terminal control software in the Multics supervisor, as well as to the code in the FNP.

Under echo negotiation, entry of new text is automatically recognized by the basic editor and the interactive driver. This case is defined as that of waiting for a command character with the current point

on the current line being at the end of it, on an already *open* line. The current buffer must not be on display in more than one window. A special call is made to the Multics supervisor whose meaning is essentially "Get a line's worth of characters, and echo them, do not wake up Emacs or return until attention is needed." Attention of Emacs (an *echo break condition*) may be any of several situations:

1. The end of the terminal line is reached. Emacs supplies the supervisor call with the computed length to the end of the screen.

2. A "non-trivial" character is typed. Emacs supplies the supervisor call with a bit-map of which characters may be echoed by the FNP: notably *not* among them are the carriage return character, the rubout character, the tab character, or any non-printing (and thus, potentially "editing") character at all.

3. An Emacs interrupt is recognized.

The occurrence of an echo break condition causes the FNP to stop echoing characters, and transmit characters (and cause wakeups) for each character typed thereafter, not echoing them, regardless of whether they are marked in the bit-map as echoable characters or not. When Emacs wakes up and receives the characters, it receives as well a count of how many were echoed: the echoed portion must be the leading substring of the characters returned. A special call is made on the redisplay to indicate that the echoed characters are now on the screen, and a special function of the basic editor inserts the echoed characters at the end of the current line. Emacs processing continues with the character that caused (or was the current input character at the time of) the echo break condition.

At the time one of these "echo negotiating" input-reading calls returns to Emacs, the FNP (and the entire Multics input system, for that terminal) is in the state it would be without echo-negotiation, namely, shipping characters one at a time, as they are typed, causing wakeups for each character (although the Multics scheduler optimizes attempted wakeups of running processes). If the successive calls to read input characters are not of the "new text entry" (echo negotiating) variety, characters will be read and processed in real-time as they are typed. It is in this state that break tables may be changed by key binding reassignments (such as occurs when switching buffers) without affecting the FNP.

When an echo-negotiating input call is made, a check is made to see if there are any input characters which have been typed, but not yet read (and thus, not yet processed) by Emacs. If so, they are returned immediately for processing, and the system remains in the character-at-a-time, non-echoing state. Automatic echoing cannot begin, for the effect of the characters read but not yet processed cannot be predicted until Emacs produces (possible) output, and makes another input call. Thus, on a slow system, Emacs is racing the typist to process characters as fast as they can be typed, and the automatic echoing state is not entered until such time that Emacs makes two successive echo-negotiating input calls during which time no input has been typed. On a slow system, the effect of echo negotiation is very noticeable: characters are echoed instantaneously until the end of the line is reached. Character echoing then stops, awaiting the loading and running of Emacs. Emacs then runs, picks up all the characters typed in the interim, performs a redisplay, outputting them all in one burst, and then instantaneous echo resumes. The number of characters for which automatic echo is not performed (some leading prefix of the typed input line) is a measure of the response time of the system. It can vary from two or three characters to a half a line in worst case, depending upon the speed of the typist. A slow typist will observe a larger percentage of automatic echo on a slow system.

The effect of the Emacs interrupt system upon echo negotiation is significant: echoing must be stopped so that console messages, updated time-of-day, or whatever asynchronous occurrence, might be displayed. The echo break condition is not detected by the communications software, but by an external occurrence in the Multics process. In this case, a special call is made to stop the system from echoing, and an acknowledgement of having stopped echoing is awaited from the FNP. The system is then in the non-echoing, character-at-a-time state, as after any echo break. Characters are then returned as usual, with an indication of how many have been echoed. The interactive driver notices the pending interrupt, and processes it as usual.

Echo negotiation is believed to be a prerequisite for any cost-effective managed video system on Multics, including tentative full-screen process output management schemes now under consideration. Future extensions to echo negotiation include rubout processing (deletion of the last typed character) and tab expansion: again, the issues that make these things at all difficult are the transparency of the human interface (consider, for example, the aforementioned feature which displays the last 50 characters typed).

Echo negotiation was conceived, developed, and released in two stages. First (January 1979), negotiated echoing was implemented in the interrupt side of the Multics terminal control software in the mainframe: while this did not reduce mainframe interrupts, it did remove the need for process loadings and wakeups on each character. The fact that synchronization (notably, the determination that there is no type-ahead, and thus echoing may begin) could be performed by the normal locking/synchronization primitives in the Multics supervisor was of fundamental importance to this implementation. The FNP was not involved in this implementation at all, and sent all characters to the mainframe one at a time during Emacs use, as before. This software was released in Multics Release 7.0.

In July 1979, the FNP communications protocols were augmented to solve the synchronization problems necessary to determine the safe start of echoing, and the stopping of echoing in response to Emacs interrupts. This moved negotiating echoing to the FNP, removing the necessity for mainframe interrupts for text entry in Emacs as well. The FNP echoing software interfaces to the previously extant interrupt-side echoing software in the exact same way that the latter interfaces to Emacs. In fact, that interface (Multics supervisor negotiated echoing to Emacs) did not change at all (by design) with the introduction of FNP echoing, and is virtually identical to the internal interface between Emacs and its input interface program. The three programs (the Emacs input interface, the Multics terminal control code in the supervisor, and the FNP software) in fact form a heirarchy of echo-negotiating input suppliers, each of which is prepared to echo characters received from the lower level, and count among those that it has echoed those echoed by the lower level. This, among the other details of the scheme, are set forth in [Echnego].

It is of passing interest that use of Emacs motivated several other changes to the Multics communications system in the period 1978-1979. In June 1978, character-at-a-time input, implemented via a "cleaned-up" version of the original patch, but no more efficient, became an official Multics feature, to allow experimentation with Emacs at MIT-Multics: Emacs use then became widespread. The communications protocol between the FNP and the mainframe had required several transactions to negotiate a place to store typed input in the mainframe: this system derives from several obsolete designs compatibility with which had been a constraint. Widespread use of Emacs, and thus character-at-a-time input, caused new, more optimal protocols to be devised and implemented in the Summer of 1978. In June 1979, the FNP input buffer strategy was totally redone, largely due to Emacs use: buffers had been collected by the communications hardware one buffer at a time per each break character. When every character became a break character, this strategy began to be very inefficient, and

under stress, failed to switch buffers with sufficient haste. A new strategy was implemented which divorces the notion of buffer-switching from that of break character.

# VII. The SUPDUP-OUTPUT ARPANET Protocol

As has been mentioned, until the introduction of character-at-a-time input as an official feature in June 1978, all usage of Multics Emacs (other than on the CISL development machine) was via the ARPANET, whose Multics implementation supported character-at-a-time input (in spite of this, a few hardy souls at the Multics Systems Release Site in Phoenix, Arizona, patched the system, with great difficulty and at great risk, in the intervening three months, to experiment).

Those desiring to experiment with the new editor would either log in to the ARPANET directly (to a *TIP*, which supports dialup connections), or from one of the local PDP-10 hosts which supported character-at-a-time interaction. The TIP [ARPANET] attempts no terminal control: characters sent by the Emacs CTLs would arrive at whatever (video) terminal was dialed to the TIP completely unprocessed. Those attempting to access MIT-Multics from the PDP-10's (running ITS), which included a large number of persons central to the development of ITS Emacs and with a strong interest in the growing editor, were faced with a different problem. ITS provides completely integrated screen management, and could not be expected to pass terminal control codes through, during TELNET connections, unmodified. Many ITS users accessed ITS through specialized locally-supplied hardware, the "Knight TV System," consisting of several dozen high-resolution raster-scan monitors, and keyboards, controlled by a central DEC PDP-11; control of these versatile, fast devices, can be accomplished only through the ITS screen management system: they have no native "control codes."

This seemingly local problem of interface between ITS and Multics was in fact indicative of a more general need, and full solution of this problem brought about the TELNET SUPDUP-OUTPUT option [RFC749] in August 1978: this option provides virtualized video support through TELNET connections. Based on the earlier SUPDUP protocol [RFC734], which is based on ITS internal buffer codes, SUPDUP-OUTPUT integrates this support into TELNET in a way that the earlier TELNET SUPDUP option [RFC736], which has no known implementations for this reason, fails to achieve.

The ITS output system operates at three levels, with three representations of output, all based upon the ASCII character set (where normal, printing characters are involved), but with three different modes of expression of control, format, and cursor position functions. Output produced by programs (*main program level*) contains, in addition to printing characters, ASCII format effectors (carriage return, tab, new line, etc.), and special escape sequences (escaped by the ASCII DLE character (Control P)) chosen for mnemonic value to effect video control and cursor position. It is into this form that the User TELNET program, by which communications to Multics is effected, that data received from the foreign host is coerced. ITS converts all such output into "internal buffer codes" (or *TDCODES*), before placing it in output buffers. The internal buffer codes have the property that all codes less than 200 octal are considered to be printing (allowing for the Stanford University Extended Character set [RFC734]), and all greater or equal are format effectors, which express context-independent cursor position and video control. In this expression, all format effectors (tabs, newlines, backspaces, etc.) have all been translated into absolute cursor positions on a video screen. (Interestingly enough, printing terminals are supported as degenerate screens). Actual conversion to device-dependent character codes and control sequences is performed at output interrupt time, from the canonical output buffer code. This strategy has the feature that output can be cleanly aborted at any point, and the interrupt side, having processed the output buffer code, is cognizant of the exact cursor position thereupon.

The SUPDUP protocol [RFC734], not to be confused with the TELNET SUPDUP option [RFC736], or the TELNET SUPDUP-OUTPUT option [RFC749], was originally developed as an inter-ITS protocol only; it is a TELNET-like protocol, to be used for communicating with one host while logged into another. Unlike TELNET [TELNET], however, SUPDUP defines a virtual, negotiated video terminal instead of the Network Virtual Terminal, a printing-terminal-like device. Hosts communicating via SUPDUP initiate their communications by exchanging a description of the user's terminal, stating its dimensions as well as what facilities it has and lacks. The server host (the one being logged into remotely, i.e., the target) thereupon converts all program output in ITS internal buffer codes, which are the official output expression code of SUPDUP. (In the case of an ITS server host, this is trivial: it simply passes its output buffers to the network. Other implementations, such as the Stanford AI Lab SUPDUP server, must convert local codes to ITS output buffer codes.) The user host (the one to which the user is physically logged in), running a "user SUPDUP program," receives all network output as ITS buffer codes, and uses it to control the local screen. Again, in the case of an ITS user host, this is trivial, it is simply placed in the local ITS output buffer directly, via a special ITS supervisor call provided for this purpose.

Those trying to use Multics Emacs at MIT-Multics from ITS hosts were immediately faced with the inadequacies of TELNET in dealing with video terminals, namely, the lack of any technique at all for describing or controlling the same. Some attempted to pass raw terminal control codes through for terminals such as the DEC VT52, but this required local "adjustment" of the ITS output system while it was being done, and failed to be transparent. This also failed to address the larger user community of the Knight TV System.

At this point, an obscure, obsolete feature of the ITS user TELNET program was revived. The Stanford University A.I. Lab (SAIL) supports a large user community using the Datamedia Model 2500 Display terminal, a moderately capable device for which Stanford has negotiated for special features with its manufacturer. Its device control codes form the standard video control code on the Stanford System: the Stanford SUPDUP server converts Datamedia 2500 codes to ITS output buffer codes. Before SAIL's creation of a SUPDUP server, those using TELNET from ITS to communicate with SAIL invoked a special mode of the ITS user TELNET program which converted Datamedia 2500 control codes into ITS main-program level codes. SAIL would treat the TELNET connection, like almost all else, as a Datamedia 2500, and the virtual Datamedia 2500 would function. In late March 1978, a CTL for the Datamedia 2500 was written, and users of Multics Emacs from ITS were told to inform Multics Emacs that they were using a Datamedia 2500, and use the Datamedia 2500 emulation mode of ITS TELNET.

This phenomenal and ingenious kludge worked tolerably well for two months before its deficiencies brought about its end. The Datamedia 2500 has a substantially smaller screen than a Knight TV: the wasted screen space due to the emulation was frustrating. So a new CTL was written for a non-existent "Large Screen Datamedia 2500," with the screen dimensions of a Knight TV, and users from Knight TV's were instructed to use this. (ITS TELNET did not limit-check the simulated Datamedia codes coming in, so this strategy worked). Even this was not enough, though. The Datamedia 2500 control language is much less powerful than the native power of the ITS output system (as expressed in the output buffer codes), lacking most notably the facility to express insertion/deletion of multiple lines. This caused multi-line insertions/deletions performed by Multics Emacs to be visibly much slower than ITS was capable of. Additional disadvantages were the need to switch in and out of "Datamedia 2500 emulation mode," and the inapplicability of the technique to any ITS-supported terminal other than a Knight TV. Switching the nature of the network connection from TELNET to SUPDUP in mid-stream

was also ruled out by these, and other considerations, this time on the ITS side.

At this time, Daniel Weinreb and the author developed the idea of embedding transmissions of ITS output buffer codes in the TELNET output stream, escaped by a special sequence. An interim protocol (known as the *PYZ protocol*) was developed and made operative in June 1978 (The name is a pun on that of the classic Czarist forgery). The Emacs user would specify the "pyz" terminal type, which selected a CTL which would express its screen control in ITS output buffer codes, escaped by a special sequence. The CTL would assume the screen dimensions and characteristics of a Knight TV. A special version of ITS user TELNET would be used, which had the added facility of watching for this special sequence, and feed the escaped data directly into the ITS output buffers in the same way that the User SUPDUP program does. The results were as expected, and this mode of usage became popular. In the next month, the PYZ protocol was augmented to negotiate the terminal characteristics in the same fashion (and format) that SUPDUP [RFC734] does. At this time, Datamedia emulation was completely obsoleted, and any ITS-supported terminal could be used via the PYZ CTL.

Mark Crispin, the maintainer of ITS User TELNET, wisely insisted that PYZ be developed into an official protocol before he would consent to standardizing the version of ITS User TELNET which supported it. Thus, in August 1978, the SUPDUP-OUTPUT protocol was adopted by the Network Information Center as TELNET option 22, and the special escapes replaced with TELNET negotiations and subnegotiations. Like all TELNET options, SUPDUP-OUTPUT can be *offered* to any server, and servers are bound to refuse if they do not support (or even have never heard of) it. Thus, upon officialization of the SUPDUP-OUTPUT protocol, and subsequent installation of the ITS User TELNET which supported it, Multics Emacs would attempt to negotiate SUPDUP-OUTPUT with any host, and automatically use a SUPDUP-OUTPUT CTL if successful. In short, the ITS User of Multics Emacs now invokes Emacs on Multics with no special action, and the entire terminal description and control is negotiated automatically.

In January 1979, a second user-side implementation of SUPDUP-OUTPUT was created, this time in a PDP-11 implementing access to MIT's CHAOS net [CHAOS] for another set of centralized-logic raster-scan terminals on the MIT campus. The CHAOS net has access to the ARPANET through specialized "gateway servers" on ITS. Some users of this terminal system use MIT-Multics through the gateway servers regularly; the SUPDUP-OUTPUT implementation on the PDP-11 allows automatic negotiated support of Multics Emacs.

The third known implementation of user SUPDUP-OUTPUT is in Multics Emacs itself, in the feature known as TELNET mode, which maintains connections to foreign hosts in Emacs windows. The (original) server implementation in Multics Emacs is the only known SUPDUP-OUTPUT server. Interestingly enough, native Multics User TELNET does not support the SUPDUP-OUTPUT option, for lack of native Multics video support at this time.

The investigations into SUPDUP led, in July 1978, to the writing of a Multics User SUPDUP, in Lisp. Built of "spare parts" from Multics Emacs, it uses the Emacs CTLs directly, character-at-a-time input (it cannot utilize echo negotiation), and, like all user SUPDUPs, performs terminal control based upon receipt of ITS output buffer codes. Almost all communication from MIT-Multics to ITS is now done with this program. The ITS user, using this program on Multics to access ITS, receives full integrated ITS screen support.

SUPDUP-OUTPUT provides a protocol via which server hosts that do not have total, integrated screen

management for all functions, but have some programs that deal in managed video, can communicate with user hosts that provide any video support at all, in a device-independent manner.

# VIII. The Place of Emacs in Multics

One of the most controversial features of Multics Emacs is the incorporation of a large number of *modes* which parallel already available function in Multics. Typical of this is the Emacs mail system, which places incoming and outgoing mail in buffers and windows, to facilitate real-time editing of the mail, paging through mail while reading it or responding to it, and automatically generating replies. There exists a complete and integrated Multics mail system, outside of Emacs, and many have validly raised the point that the existence of another one, inside Emacs, nowhere as complete, is questionable at best.

However, the task of mail composition seems to overlap so largely with the task of text editing, that integration with a text editor seems appealing. In the standard Multics mail system, a sharp distinction is made between "inputting" mail and "editing" mail. The use of multiple windows to read and reply to mail, with the ability to page back and forth, is so natural that some have wanted to learn to use Emacs for this reason alone. Certainly, if Multics had integrated video management (which is at this time under serious design consideration), the mail system could use it (and will) to advantage: indeed, the Emacs mail system is indeed a way of getting "video-managed mail" if nothing else. However, the large percentage of the mail-composing/reading task which is editing mandates that the most potent editing technology available be used, and this is Emacs. Emacs seems a more likely candidate to contain a mail system than the mail system to contain an Emacs, so this is the way it was done. (On ITS, an Emacs-imbedded mail system exists as well).

The unique nature of the Multics process environment, specifically, the ability to call any procedure or subsystem known to Multics, if proper interfaces exist, allow a wide panorama of function to be subsumed into Emacs, and experimentation with video interfaces to Multics function to be performed. Creating function via the Emacs extension language and calling of external Multics routines begets utility without having to build an environment from ground up, and buys video management for free (by virtue of the automatic redisplay).

Prototypical of many "special-purpose" Emacs modes is the *directory editor*, *DIRED*, which exists in both the ITS and Multics Emaces. The user, inside Emacs, invokes the directory editor via a sequence of command characters. A display listing all files in the storage system directory to be "edited" is placed in a buffer (and thus, by virtue of the redisplay, displayed on the screen). Normal Emacs commands can be used to position to any line (each line describes one file of the directory) of the display. In this mode, no commands which would cause the buffer to be modified may be issued. However, commands such as "delete this file" and "show me the contents of this file" are available on keystrokes. Thus, the user positions around the display of the directory listing, examines files, and marks them for deletion (they are actually deleted when the directory editor is exited). The user never has to specify file names, and sees a "large picture" of what files are in the directory at all times. This "menu"-type interface is typical of many advanced video systems [PARC].

Menu-type sub-editors are provided in Multics Emacs to "edit" buffers (examine and destroy unneeded ones) and windows (move around their boundaries, set their sizes) as well.

Another class of special-purpose modes invokes large-scale Multics subsystems from within Emacs, and processes their output in a useful way. In Lisp mode, a single keystroke invokes the Multics Lisp

compiler upon the function at which the cursor is pointing, and upon its completion, incorporates the object program into the running MacLisp environment, and displays the compiler's diagnostics on the screen. In PL/I and FORTRAN modes, the same keystroke invokes the appropriate compiler upon the source program being edited. The compiler's diagnostics are placed in a buffer in a second window, and are analyzed to identify the source lines flagged as in error by the compiler. A dedicated command (keystroke) in PL/I and FORTRAN modes moves the cursor in the source program to the "next line flagged as in error by the compiler", and moves the "current `point`" in the diagnostics buffer to the next diagnostic, such that it is displayed in the diagnostics window automatically. The "pointers" to the source lines are kept as *marks*, and are thus valid through the new editing of the source program. In this way, multiple windows are used to "reply to" source-program errors in the same way that mail responses are generated.

The interactive message processor in Multics Emacs (already described under the discussion of the Emacs Interrupt System) creates buffers associated with senders of interactive messages. These buffers have their key bindings so set up that typing lines into them will send those lines as interactive messages to the associated user. As messages sent by the other user appear at the end of the buffer automatically, a "conversation" can be held with another user simply by "going to" such a "message buffer." Multiple message buffers (like any other buffers) can be displayed on the screen in multiple windows, and thus several conversations can sometimes be seen scrolling simultaneously, automatically, on a Multics Emacs screen. This facility goes so far as to allow automatic routing and response to interactive messages coming from foreign network sites.

A third type of special mode is provided by User TELNET and User SUPDUP modes. In these modes, communication with a foreign ARPANET host is established, and a dedicated buffer (and usually window) becomes a virtual video terminal of the foreign system. All text typed into the TELNET or SUPDUP buffer goes to the foreign system. In TELNET mode, the foreign system is assumed not to have integrated screen support or real-time line editing (by virtue of the TELNET protocol), and all the power of Emacs editing may be applied to input before it is "sent" by a carriage-return. In SUPDUP mode, screen management is performed by the foreign system, and received network output is interpreted as ITS output buffer codes [RFC734] to position around the SUPDUP mode buffer. These modes are indicative of a desire of Emacs users to stay within Emacs during the entire duration of their Multics login session, "running their processes from Emacs." Again, the desirability of these features would be questionable if a Multics process could have "a TELNET" and "an Emacs" running at once, activating either when needed. The Multics stack architecture, however, makes this extremely difficult if not impossible. The "coroutine" architecture of Emacs buffers is a better model of multitasking in a process, and provides an easy way to divide the screen between these "tasks" as well.

Another special-purpose mode of interest is Lisp Debug mode, intended for debugging Emacs extensions. A special-purpose buffer is set up in which each line typed is evaluated as a Lisp *form* when carriage-return is entered. The result of the evaluation is placed in the buffer as well, resulting in a *transcript buffer*, like that of console-message mode. Lisp Debug mode allows for the setting of breakpoints in extension code being debugged: when a breakpoint is entered, the Lisp Debug mode buffer is entered, with an appropriate message. Variables may be examined and set via normal Lisp forms typed into this buffer. The Lisp evaluator's stack (it is best to debug functions in interpreted form) may be traced. A special command in Lisp Debug mode returns from the breakpoint.

Lisp Debug mode uses multiple windows to advantage, too. It is usual to use three windows when using it; one displays the source code for the extension being debugged, one displays the text upon which the

extension is operating, and the third the Lisp Debug mode buffer. When a breakpoint is entered, commands in Lisp Debug mode are available to display the breakpoint source in the first window, and show the position of the "current `point`" in the buffer being operated upon by moving the cursor there in the appropriate window. Thus, the "current `point`" may be inspected and manipulated (by issuing normal Emacs commands in the buffer being operated upon) during a breakpoint, just as a variable in most debuggers. One can thus "step through" extensions, via breakpoints, watching their action upon text buffers (automatically redisplayed at breakpoint time, like all else) as they proceed. Halbert [DIDL] has also implemented a Lisp-coded display-oriented debugger which shares many concepts.

# IX. Experience and Conclusions

In the year and a half since its inception, Multics Emacs has grown from a toy Lisp program to a multi-thousand line subsystem encompassing widely diverse Multics facilities and used across the country. It has inspired a wide variety of reaction, which is in many ways telling about the state of the computer marketplace.

In most ways, Multics Emacs shares the ITS Emacs experience: novice and experienced users find Emacs easy to learn and to use, and those who use video terminals rarely revert to earlier editing habits. People seem to become productive and proficient with Emacs in less time than with the "conventional" editors. Skilled programmers build extension subsystems to accomplish sophisticated tasks, and libraries of personal Emacs extensions abound.

The impact of Emacs upon Multics, however, is quite unparalleled in the ITS experience. Cognoscenti at once recognized the ostensible similarity between Emacs and stand-alone word-processing systems, and attempted to identify Emacs as an integral part of Multics Word-Processing. Ways were sought to support dozens, or hundreds of Emacs users, in vain attempt to approach the economy of the stand-alone word-processors. The power of a mainframe real-time editor, however, is not in its non-existent ability to compete economically with dedicated minicomputer systems in a word-processing environment, but in its extensibility, which allows it to perform more highly specialized and sophisticated editing tasks (*cf.*, the PL/I-FORTRAN error diagnostic mode above) than any minicomputer system is capable of.

People at the sites where Multics Emacs is used appear to use it regardless of the substantial resource cost of doing so; the redisplay-based real-time video editor is innately expensive: people deal with that reality, and often use Emacs even on heavily-loaded systems. On many systems, users do not directly pay for resources consumed, which makes expensive tools attractive.

The emergence of Multics Emacs has sensitized Multics users to the advantages of integrated video support, and real-time line editing. The widespread conversion of time-sharing users to video terminals has left the printing-terminal oriented Multics interface far behind, and Emacs has let users see how video terminals can be managed intelligently. As a result, there is now substantial agitation for integrated video support in the Multics terminal support and communications areas.

The emergence of Multics Emacs also seems to have put the final nail in the coffin of the Multics Editor issue. No suggestions about new printing-terminal editors have surfaced since Emacs came about.

Multics Emacs has been responsible for renewed interest in Lisp: many people wanting to write Emacs extensions have pursued Lisp (some knew *no* other programming language), and have rapidly become enamored of it. Multics Emacs is a tremendously potent tool for the creation and debugging of Lisp

programs, taking all of the pain out of indentation, parenthesis-balancing, and similar mechanizable tasks.

The power of Multics Emacs as a tool for developing itself, i.e., extensions, cannot be overstated. The line-by-line, function-by-function incremental input and debugging of code facilitated by Lisp mode and the active MacLisp environment allow code to be produced and debugged in seconds, and finely tuned and reiterated in the same way that a sculptor polishes and examines his or her work. There is no interactive program development facility on Multics or ITS anything like Emacs developing its own extensions: the power of this facility has been directly responsible for the large number and wide variety of extensions that have come about.

Multics Emacs has also provided a testbed for experimentation with alternative Multics user interfaces, and highly customized environments. It could be validly stated that the user interface of Multics does not extend well to video terminals, and needs be almost completely overhauled for effective video terminal utilization: the mail system is a case in point. Multics Emacs has provided a path for experimentation with "entire alternative interfaces" to Multics.

Multics Emacs has provided a subject matter of interesting discourse between the Multics Development Community and other researchers at MIT, Stanford, and elsewhere working on similar related issues. In addition to a valuable infusion of new and radically different ideas into Multics, there has been a give-and-take with those investigating similar interfaces [Stallman] [DLWThesis] [SINE] [Schiller] during its development. Multics Emacs in fact "grew up" with these related editors, which were an active, state-of-the-art research topic at the time.

The choice of Lisp was a bold one: it alone has been responsible for the rapid development of tremendous function in Multics Emacs. It is clear that an implementation in PL/I, with no concessions to elegance or modifiability, like one in assembler language, would support a larger number of users at smaller cost, but none of this function would have been developed.

The choice of Emacs as an interface was an extremely fortunate one: all subsequent observation has indicated that the intuitive smoothness of the Emacs interface (by and large) is directly responsible for its ease of learning, and its conceptual advantage over the conventional editors.

# X. Acknowledgements

The author is deeply indebted to a large number of people for assistance and encouragement during the prehistory and development of Multics Emacs. I would like to thank:

William York, Gary Palter, and Richard Lamson, the other maintainers and developers of Multics Emacs, each responsible for numerous features, fixes, and enhancements, and infinite work.

Richard Stallman, for developing the entire concept of Emacs, and offering fervid and enthusiastic support from the first day.

Honeywell Information Systems, Inc., specifically Charlie Clingen, John Gintell, and Steve Webber, for the foresight to allow resources to be committed to this project, and for developing it into a Honeywell product.

Dan Weinreb and Dave Moon, for all natures of assistance, information, encouragement, and support too numerous to mention.

Larry Johnson, Jerry Stern, and Robert Coren, the maintainers/developers of the Multics communications software.

Earl Killian, Eugene Ciccarelli, and Bruce Edwards for early guidance and continued support, and their vast experience in terminal support and real-time display software.

Paul Schauble, in Arizona, for his contributions of FORTRAN mode and the FORTRAN/PL/I diagnostic scan mode.

Roger Roach and the staff of the MIT Information Processing Center for allowing experimentation with character-at-a-time real-time Lisp-coded editors on their service system, and allowing an Emacs user community to grow there.

The MIT AI Lab, for allowing me to use their system, and become familiar with ITS.

Lee Parks, Charles Frankston, Carl Hoffman, Lindsey Spratt, Suzanne Krupp, Margaret Minsky, Gerry Sussman, and a large number of other people for all sorts of help, encouragement, and ideas along the way.

# Bibliography

[AG94]
*Multics PL/I Language Specification*, Order #AG94, Honeywell Information Systems, Inc.

[ARPANET]
*ARPANET Protocol Handbook*, Network Information Center.

[Bell QED]
Bell Telephone Labs: QED.

[CHAOS]
MIT AI Lab/Lab for Computer Science: CHAOS net.

[CTSS]
Crisman, Ed., et al., *CTSS Programmer's Guide*, MIT, 196x.

[Chineual]
Weinreb & Moon, *The Lisp Machine Manual*, MIT, 1979

[DIDL]
Halbert, D. *A Symbolic Debugger &c&c* ... MIT S.B. Thesis.

[DLWThesis]
Weinreb, Daniel L., MIT S.B. Thesis on ZWEI editor

[EDOC]
Documentation on E Editor (E.DOC[ALS,UP] (??))

[Echnego]
Greenberg, Bernard S. *The Echo Negotiation Papers*, Multics Technical Bullet 418, Honeywell, Cambridge, Mass, July, 1979.

[GCPAPER]
Ancient CACM article on what became Multics Lisp GC, by Fenichel, Yochelson, etc.

[ITSDOC]
Eastlake, et al., AI Memo xxx, *ITS User's Manual*
[LCP]
Greenberg, Bernard S., *The Multics MacLisp Compiler, the Basic Hackery*, unpublished paper, Honeywell, Dec. 1977
[Lispbook]
Greenberg, Bernard S., *Notes on the Programming Language Lisp*, MIT SIPB 1976, 1978.
[MPM]
*Multics Programmer's Manual*, AK.. ..... ....
[Moonual]
Moon, David A., *The MacLisp Reference Manual*, MIT, 1974.
[Multics]
Corbato/Clingen/Daley paper (click here for Multics info.)
[PARC]
Teitelman, *A Display Programmer's Apprentice* (??) Xerox PARC.
[QUG]
Honeywell, QEDX user's guide, in preparation.
[RFC734]
Crispin, M., "The SUPDUP Protocol," Network Document NIC-41953, RFC 734.
[RFC736]
Crispin, M., "The SUPDUP TELNET Option," Network Document NIC-42213, RFC 736.
[RFC749]
Greenberg, B., "The SUPDUP-OUTPUT TELNET Option", Network Document NIC-45499, RFC 749.
[SINE]
Anderson, Owen T. ("Ted"), MIT S.B. Thesis on SINE editor.
[SWG]
*Multics Subsystem Writer's Guide*, Order #AK92, Honeywell Information Systems, Inc.
[Schiller]
Schiller, Jeffrey I., MIT S.B. Thesis on TORES Editor.
[Stallman]
Stallman, Richard M., *Emacs, the Customizable, Extensible Display Editor*, AI Memo XXX, MIT AI Lab, 1979
[TECDOC]
Online documentation for TECO, MIT AI system.
[TELNET]
TELNET Protocol, Network Document #XXXXX.
[TMACS]
???
[Tools]
*Multics System Programming Tools*, Order #AZ03, Honeywell Information Systems, Inc.

---

# Appendices

## Appendix A. The Redisplay

The Multics Emacs redisplay has generated some excitement, for it was designed with the hindsight of several other redisplays, and optimizes visibly well, especially at low speeds. As there are not many detailed analyses of redisplay algorithms in the literature, it is instructive and valuable to consider the algorithm of the Multics Emacs redisplay in detail.

The redisplay was intended for use in, at worst, very low speed (300 baud) environments, and to take advantage of terminal editing features (line and character insertion and deletion) as much as possible. Multics does not (yet) provide the ability to synchronize with the termination of output, or a way of telling whether output is actually in progress (this seems to be a common feature of multi-node communication processing networks), and because of this, Multics Emacs appears to often generate "extra" redisplay (in the sense of "screen updating") at low speeds. (Compare [Stallman], wherein ITS Emacs is described as being able to stop redisplays as new input is typed.) Other than inadequacies resulting from this, the Multics Emacs redisplay seems to fulfill well its goal of optimal redisplay at low speeds.

The basic function and data structure, and terminology relating thereto, of the redisplay, are that as given in Section IV above. The data-structure and terminology of the window management system is as given in Section V.

---

The redisplay is invoked by the interactive driver after each input character has been processed by a call to the basic editor or an extension. The call is suppressed if there is already input "read-ahead" from the Multics Supervisor. The redisplay is given no explicit arguments, and returns no value. The basic task of the redisplay is twofold: to update the screen contents such that all windows correctly display the contents of the buffers on display in them, and to then position the cursor to the point in the selected window corresponding to the "current `point`" in the current buffer. The redisplay data structure, at the time the redisplay is invoked, describes the final contents of the screen, both logically (what Editorlines) and physically (what characters), at the time of the last redisplay. The secondary goal of the redisplay is to utilize this information to *transform* each window to display its correct contents.

The first phase of the redisplay (update all windows to their correct contents) is driven by iteration over the array of window information alluded to in Section V. If the layout (i.e., number, extent, and content) of windows has changed since the last redisplay, the redisplay is not aware of it, and simply considers the current contents of the current windows, including lines which may have been from other windows, and lines of dashes used to divide windows. (These window-dividing lines are actually implemented in the lowest-level window-description array as windows in themselves: this is possible because of the full-width nature of windows, and the full line of the terminal needed to "draw" them). "Incorrect" lines will be changed regardless of their origin.

For each current window, the task is, as stated above, to compute the new contents, and then update the screen to reflect it. At the end of this processing, the **Displaylines** representing the new screen contents will be placed into the correct elements of the **screen** array for the next redisplay (the screen array is indexed by lines, not windows, and thus, changes in the window layout between redisplays can be ignored). The computation of the new window content itself consists of two phases, which are implemented as one pass, but are logically distinct: a choice of what **Editorline**s are to be displayed in the window must be made, and the **Displaylines** representing them (including the printed representation)

must be computed.

The choice of **Editorline**s is based upon one criterion: the **Editorline** which is the basic editor's "current line" must appear in the window, in its entirety (if at all possible). (For other than the selected window, the line of the "window `point`" is used equivalently: see Section V.) Most of the time, the current line is already on the screen. When it is not, a new choice of lines must be made. Thus, a three-pass algorithm is used to choose lines for the new window. Each pass fills the appropriate (for this window) elements of the `newscreen` array with freshly-computed **Displayline**s, representing the printed representation of some extent of the appropriate buffer from some "tentative window starting point" on down, until a full window's worth of **Displayline**s have been computed. As each pass computes a tentative window-content, the screen position assigned to the current **Editorline** (if it was encountered at all) is remembered. A pass "succeeds" if and only if the current line was in fact encountered, in which case no further passes are made. Failure of a pass means that the tentative window contents do not include the current line. The position so remembered will be used by the final phase of the entire redisplay to position the cursor in the selected window.

The first pass attempts to use the old choice of **Editorline**s, which handles the vast bulk of cases. It takes the **Editorline** which is the starting line of the current window contents (as determined from the **Displayline** of the first element of this window's extent of the screen array) as a starting point. Now this **Editorline** may not even exist: it may have been deleted by the basic editor since the last redisplay! To account for this, the basic editor replaces the **Linecontent**s of deleted lines with a unique object which the redisplay recognizes in this case. If this object is found to be the **Linecontent** of the first **Editorline** of the old screen, the second is used, and so on. Like the other passes, it computes the **Displayline** for each **Editorline**, following the doubly-linked list of the buffer structure to get the next **Editorline**, and fills the screen array with the **Displaylines**.

The second pass, which is invoked when the first pass failed to find the current line in window contents based upon the previous window contents, attempts to choose a window content that places the current line in the center of the window. The second pass *measures* (predicts the vertical screen-extent of) **Editorline**s, going back through the buffer via the doubly-linked list, until one half a window's worth of lines have been passed. The line on which it finds itself is tentatively chosen as the window's starting line, and a tentative window content computed.

The third pass is invoked when the second pass has failed to find the current line. This can only occur in very rare cases of extremely long lines during normal redisplay. However, the window redisplay procedure is also invoked at times on *demand* (for user commands such as "display next window's worth of text," which are quite common and useful), and at these times, the third pass is "opted for by request," to *explicitly* choose a window content which has some given starting point. The third pass is *guaranteed* to find the current line.

The computations of **Displayline**s are made storage- and time-efficient by an optimization which is based upon the fact that if a line (more precisely, an **Editorline**) has not changed, its printed representation (more precisely, the **Displayline**s comprising its representation in the redisplay data structure) cannot have changed, and thus, the **Displayline**s, if they can be found, need not be recomputed. An array, **old-Linecontents**, is maintained, with an element per screen line. At the end ofthe window update process, the **Linecontent** of each **Editorline** on display in the window is stored (specifically, at the machine level, a pointer to the Lisp string or **Filecons** which is the **Linecontent**) in

the corresponding element of **old-Linecontents**. When a **Displayline** is to be computed, the old window content is scanned for an instance of the **Editorline** whose (new) **Displayline** is being computed. At the machine level, this is a comparison of pointers. The deliberate "identity" of lines given by the buffer representation used in Multics Emacs is the basis of this technique. If the **Editorline** is not found, a **Displayline** is computed and constructed. If the **Editorline** is found, a check is made to see if it still has the *same* (i.e., the identical instance of a Lisp object, that is to say, pointers to it are equal) **Linecontent** as that saved in the corresponding element of the **old-Linecontents** array. If so, the old **Displayline** is still valid, and no **Displayline** computation or generation need be performed, and the old **Displayline** is used. This technique relies upon the fact that if an **Editorline** was modified, it *must* be given a different **Linecontent** than it had before, and Lisp storage is never re-used, so the saved pointer in the **old-Linecontents** array can be used to determine unambiguously whether or not an **Editorline** has changed since the last redisplay. The old **Linecontent** of the current line is never stored (a unique object is stored instead), to force full **Displayline** computation of the current line. This is required by optimizations that will be described subsequently.

Once the window's new contents have been determined, the task of comparative update begins. A linear scan through the **Displayline**s of the new window contents is made, maintaining a pointer performing a linear pass down the old window contents in parallel. As each new **Displayline** is processed, a scan is made down the remaining portion the old window looking for a **Displayline** describing the same **Editorline** as the current new **Displayline**. If it is found further down the old window than the current pointer along the old window, the CTL is requested to delete the appropriate number of lines from the terminal's screen. (If insertion and deletion of lines is not available, **Displayline**s are simply compared and updated one-for-one, via the line update procedure described below). If it is not found, the **Editorline** of the **Displayline** at the current pointer of the *old* window is sought among the remaining **Displayline**s of the new window, and if found, the CTL is asked to *insert* lines (open up new space) on the terminal's screen. Once these manipulations have been performed (and pointers and indices updated to reflect lines deleted or new blank lines on the screen), a **Displayline** representing the most appropriate line on the screen to be comparatively updated (redisplayed) to be the current **Displayline** will be in hand, and the screen will have been reorganized to actually place that line where the new line is to be displayed. If neither line can be found on the other screen image (the old on the new or the new on the old), the current **Displayline** is redisplayed with respect to whatever happens to be on the screen at this point.

Insertion and deletion of lines is used to open up or close up space in windows; for every insertion, a deletion must be performed first at the bottom of the window, and for every deletion, an insertion of blank lines at the bottom of the window must be performed subsequently. "Region scrolling" features, available on the HDS Concept 100 and DEC VT100 terminals, provide this function directly, and a much more pleasing behavior of Multics Emacs is a result.

After insertion and deletion has been performed, old **Displayline**s are redisplayed into new **Displayline**s via the line-update procedure. Two trivial checks are made for identical lines: a pointer comparison on the corresponding **Linedata**s (which will succeed if the optimization of **Linecontents** above was employed), and that failing, a string comparison. An empty, or non-existent, line, at this point, is considered to be a line of as many blanks as the "other" (i.e., old vs. new) line. The content of blanks in fact represents what is to be put (for new lines, or what is, for old lines) on the screen. The "length of the blank line" is chosen for the benefit of the comparative update algorithm: the obvious alternative of "one screen line's worth of blanks" is less well-suited because of the availability of terminal features (i.e.,

"clear to end of line") which special-case blank ends-of-lines.

The **Linedata**s of the old and new **Displayline**s are now compared character-for-character from either end, to find the point where the differ in each direction. For example, old and new lines (respectively) of

```
able to leap tall buildings in a single bound
able to find tall persons within their bound
```

will be found to have eight characters in common on the left ("able to "), and six on the right (" bound"). This computation is not cheap, and hardware help would be of great value here. The two lines are now partitioned into a *left common area*, a *difference area*, and a *right common area*. Note that the presence of "tall" in the difference area is not recognized. This model of line changes is intended to reflect common user behavior in a text-editing environment: for some other types of redisplays (e.g., one which updated screens showing system status, main memory maps, etc.), this might not be optimal, but in a text-editing environment, truly wonderful redisplays have been achieved through this model.

For equal-length difference areas, the new difference area can be rewritten in place, for any device that does not overstrike. Thus, commands such as "Uppercase the word at which the cursor points" have immediate, visible, and local effect, creating minimal redisplay. For overstriking devices, this cannot be done, and equal-length difference areas is not a special case.

For non-equal-length difference areas, insertion or deletion of character positions must be performed to preserve the right common area, when there is one. The common length of the difference areas is overwritten (for non-overstriking devices), and the remaining length is deleted out (if the old is longer), or inserted (if the new is longer).

For zero-length right common areas, right common areas consisting entirely of blanks, or when using terminals that do not support insertion and deletion of characters, the old line (now on the screen) is cleared from the end of the left common region to the end (if indeed it was at least this long), and the new difference area (if non-blank) is printed (non-blank right common is, at this point, considered as part of the difference area for non-insert-delete-character terminals: note that non-blank right common *is* significant for such terminals in the equal-length difference-area case.)

To optimize for very weak terminals that do not support clearing to the end of the line (such as "glass teletypes" with no cursor addressing), clearing to the end of the line is delayed as long as possible, rewriting the new difference area first. This allows only what remains on the line to the right of the new difference area to be be cleared out by overwriting with blanks.

A very potent optimization is made for overlength (multi-**Displayline**) lines which have had characters inserted or deleted in the middle of their non-final **Displayline**s. Consider the old/new pairs of lines:

```
(old):          I have seen light,
                \cand rejoiced.

(new):          I have seen a ligh
                \ct, and rejoiced.
```

(The **\c** is the Multics conventional character sequence for continuation lines) Consider the redisplay of the first **Displayline** "I have seen light," vs. "I have seen a ligh"). The left common area is the text "I have seen ". The difference area is the rest of either line, there is no right common area. Yet, a natural

operation in our model (insertion of "a ") has occurred, and a large amount of redisplay (rewriting the rest of the line) will occur if we take no further action. We would like to be able to take advantage of character insertion here. Thus, to detect this case, a string search (via hardware) is made for a suffix substring (an "end" of the string) of the old difference area which appears intact in the new difference area (e.g., " ligh"), and vice versa (to detect the deletion case). This is done only for two **Linedata**s as long as the width of the screen, with no right common at all. If either check succeeds, the difference area is "scrolled horizontally" using insert/delete character features of the terminal. (Sometimes the "wrong" check (i.e., the unexpected), succeeds prior to the other check, and very amusing (although completely correct) redisplays result).

All text displayed by Multics Emacs is ultimately put on the screen by the line-update procedure. Indeed, even the display of completely new text, e.g., a file just read in, upon a blank screen, is seen as a line update against a line of blanks, with a left common equal to the amount of leading whitespace (indentation) on the line, and the rest of the line a difference area, with no right common. Thus, the printing of the difference area, in the zero-right-common case, is extremely important, statistically, and is responsible for the majority of characters output by Emacs (exclusive of echo-negotiated input). An optimization exists (for other than weak terminals with no clear-to-end-of-line) to take advantage of cursor addressing for avoiding printing of whitespace when text is output by the line-update procedure in this case. The text to be so displayed is scanned for whitespace: any significant run of whitespace is displayed by clearing to the end of the line (this is only done once, if at all) and positioning to the start of the new non-blank text.

The first and most important optimization of the Redisplay to be implemented was in some sense the ancestor of echo negotiation, and should be implemented by anyone considering designing a redisplay. As in echo negotiation, the goal was to optimize the insertion of "trivial" characters (see Section VI) at the end of the current line. The objective of this optimization is to avoid the substantial time and storage expense of the full computations of the redisplay, even with all of the optimizations described, for this tremendously important case. The implementation thereof involves one of very, very few instances of the basic editor's awareness of the redisplay.

To allow the redisplay to detect this case, a flag, reset by redisplay each time it is invoked, is set by the basic editor any time any modification is made to any **Editorline** at all, except insertion of characters at the end of the unique **Editorline** which was the current line at the time of the last redisplay. Any demand reorganization of the screen (e.g., redefinition or reorganization of windows) sets this flag as well. The redisplay sets a variable, visible to the basic editor, to that **Editorline**, for this purpose. If, at the time redisplay is invoked, that line is still the current line, and the flag has not been set, and the current point position on the current line is at its end, the entire redisplay may be performed by simply typing (i.e., sending to the terminal, via the CTL) the characters by which the current line has been extended since the last redisplay. (Actually, the presence of non-trivial (e.g., TAB) characters in the line extension, and physical line overflow must be checked for as well, and cause this optimization not to be used if present.) The old value of the location of the current point on the current line is maintained by the redisplay so that the line extension string may be identified.

This optimization, known as a *quick redisplay*, subsumes the end-redisplay positioning of the cursor and updating of the **old-Linecontents** array (which, as had been mentioned, avoids allowing the current line to be optimized). It does not, however, subsume updating the screen array. However, the only possible object or part of an object which could have changed is the **Linedata** of the **Displayline** (that where the cursor was left by the last redisplay, whether it was quick or otherwise, which always describes the

current (**Editor**)**line**). It is critical to have a correct **Linedata** for the current line, for even though the quick redisplay does not use its contents, the next redisplay might not be a quick one, and it will be needed. Construction of a new Lisp string for that **Linedata** for each quick redisplay would (exclusive of echo negotiation), for a line of *N* characters, generate *N* strings during *N* quick redisplays, creating *N**(*N*+1)/2 (order of *N***2) characters of storage. To avoid this storage generation, the **Linedata** of the "current **Displayline**" is maintained as a "rplacable string" (See Section IV). Quick redisplays simply update its contents. The final action of normal (non-quick) redisplays is to instate this string as the **Linedata** of the current **Displayline**, if it is not already so, and allocating real storage for the **Linedata** of the last line in this state (if not the same line).

Upon return from an echo-negotiated reading of characters, the redisplay is invoked to update its screen image. Note that by definition of the valid circumstances for echo negotiation and quick redisplay, a quick redisplay would result. In fact, a special flag is set (suppressing the actual output of characters by the redisplay, as echo negotiation output them), and the redisplay is invoked. If, in fact, any redisplay other than a quick redisplay results, an internal error has occurred. Note that the normal redisplay and the quick redisplay can be considered as even two more levels of the echoing hierarchy described in Section VI.

One remaining optimization similar to the last remains. If the flag used to deselect quick redisplays has not been set, and no modification *at all* has been performed upon any line (as determined by another flag reset at redisplay time and known to the basic editor), but the current line is *not* the same line as at the end of the last redisplay (a quick redisplay will therefore *not* occur in this case), the redisplay will conclude that all that has happened is that the user has "moved the cursor around" on the current line or in the current buffer. If the current line can be found in the current selected window, the entire window update phase of redisplay is skipped, proceeding directly to the final cursor positioning. This important optimization was much more important yet before the **old-Linecontent** optimization was implemented.

# Appendix B. The Extension Language

Multics Emacs extensions, whether part of the standard editor, loadable libraries, or user-written, are written in Multics MacLisp, augmented by a set of Lisp macros provided as a lexically-includable program fragment with the system. Extensions are written in an *environment* consisting of the native MacLisp functions (other than I/O), functions in the basic editor and standard extensions, and occasionally the redisplay. The basic editor functions provide the ability to manipulate the current point, and the buffers, and inspect and change the contents of lines and buffers. Lisp macros are provided for syntactic sugaring of commonly used syntactic cliches, such as "create a temporary variable, assign a mark at the current point to it, perform some code, and free the mark," as well as to augment the basic expressive power of MacLisp.

The writer of extensions creates MacLisp functions via *defun*, the standard MacLisp function definition facility. In addition to invoking functions in the extension environment, these functions may invoke each other (as may any Lisp functions), or be "connected" to keys, so that they will be invoked automatically by the interactive driver when selected keys are struck.

Extensions use as data strings, integers, buffer-names, and marks (see Section V). The basic Lisp data types (symbols and lists (implemented via *cons*es)) are only occasionally used. In fact, reasonably expert extension writing has been accomplished by persons completely ignorant of fundamental Lisp data

objects types. The fact that marks are implemented as lower-level Lisp objects is transparent and irrelevant to the Extension writer: he or she is not allowed, and never has reason, to "decompose" them; such is the elegant nature of the Lisp object abstraction. The extension writer has no knowledge of or dealings with **Editorline**s, **Displayline**s, **Linecontent**s, or any of the other elements of the basic editor or redisplay data structures mentioned in preceding sections.

The most useful class of function used in extensions are those which are already capable of being invoked by keys by the interactive driver. For instance, **forward-word** is very commonly used in editing to position the cursor past the current word, which is how the Emacs user conceptualizes "what Escape-F does" (Escape-F being the two-key sequence standardly used to invoke this common command). The extension writer, on the other hand, conceptualizes the **forward-word** function as moving the current buffer point to beyond the current word. Using these functions in extension functions is a valuable technique: the extension programmer can always experiment with the function to be used by invoking it in the normal interactive (i.e., via keystroke) way to determine details of its behavior.

Here is a simple example of an extension function based upon commands normally available through the keyboard. Its name is "**bracket word**", and it places the word at which the cursor points in angle brackets:

```
(defun bracket-word ()
       (forward-word)
       (insert-string ">")
       (backward-word)
       (insert-string "<"))
```

The empty parentheses, (), signify, in this context, a function which has no formal parameters. The function **insert-string** has the same effect as the interactive user typing a sequence of self-inserting (trivial, printing) characters. The invocations of **forward-word** and **backward-word** position the current point prior to the insertions of the character strings. The end result of running this function would be the same as if the user had typed Escape-F (which invokes **forward-word**), a right-angle-bracket, Escape-B (which invokes **backward-word**), and a left angle bracket. The net result on the buffer (and the screen) is the same. However, the intermediate states which would be visible to the user typing the above sequence will not be visible on the screen when this extension is run, only the final state will be. This is because the interactive driver invokes the redisplay after each command character is typed, but this function (as is visible by inspection) does not invoke the redisplay, it invokes only what it is seen to invoke.

The most common extension environment macro is **save-excursion**, which is used to remember the location of the current `point`, and restore it after the execution of the included code within the macro. For example, the following extension function places a star at the beginning of the current line, but leaves the cursor at the same place at the current line: (Bear in mind that the position is remembered via a mark, which is relocated automatically as the buffer text changes)

```
(defun put-star-at-beginning-of-line
   ()
      (save-excursion
        (go-to-beginning-of-line)
        (insert-string "*")))
```

The **save-excursion** macro encompassing the invocations of **go-to-beginning-of-line** and **insert-string**

ensure that the current point will restored after these functions run. Another similar macro, **save-excursion-buffer**, is used to restore the selection of buffer during its dynamic scope. As switching out of a buffer saves the location of the current point within that buffer, **save-excursion-buffer** subsumes the task of saving the point within that buffer.

Another set of very common macros in extension writing are those dealing with marks, providing for the creation thereof, and freeing at the end of the contained code. Freeing of marks is very important, as the length of the basic editor's mark list limits the speed of line-opening. The macro **with-mark** names a variable to which a mark is assigned at execution time: that mark will denote the point in the buffer which is current at the time the code contained in the macro begins execution. The following extension function which deletes two words forward from the current `point`:

```
(defun delete-two-words-forward ()
       (with-mark here
         (forward-word)
         (forward-word)
         (wipe-point-mark here)))
```

When **delete-two-words-forward** is invoked, a mark designating the current `point` in the buffer is created, and assigned to the local variable named **here** (these are all artifacts of the **with-mark** macro). The two calls to **forward-word** are then executed, presumably moving the buffer `point` (but not the saved mark) two words forward in the buffer, and then **wipe-point-mark** is invoked, passing that mark as an argument. **wipe-point-mark** deletes all text between the current buffer `point` and the `point` designated by the mark (saving it, incidentally, for possible user recovery). At the end of execution of **delete-two-words-forward**, the mark created by the macro is freed.

Another class of Emacs extension environment macros are those used to supplement (or reimplement) features in MacLisp thought to be inadequate, either for learning purposes, or ill-tuned to the extension environment. For example, the extension documentation teaches the use of the **if** macro as opposed to the native MacLisp **cond** as the fundamental conditional construct. **if** is much simpler and straightforward, suffices for almost all cases, and is similar to the conditional construct in almost all languages other than Lisp. The native MacLisp **cond** is much more general and powerful, but this power is not often needed, and seems to have presented a stumbling block to those learning Lisp. Another macro of this class is **do-forever**, and its exit form, **stop-doing**. The native MacLisp **do** has two forms, one like the FORTRAN **do**, and the other a powerful multi-variable generalization of this. Most often, the extension writer wants to iterate not over integer variables, but over buffer lines or characters: the iteration variable is thus the global editor state, and the need to specify or deal with variables which are almost never needed is undesirable. **if** and **do-forever** are illustrated by the following extension function, which either finds the first blank line of the buffer or complains if there are none:

```
(defun find-first-blank-line ()
     (go-to-beginning-of-buffer)
     (do-forever
       (if (line-is-blank)
           (stop-doing))
       (if (lastlinep)
           (display-error
             "No blank lines!"))
       (next-line)))
```

The form **(stop-doing)**, of executed, causes control to exit the **do-forever** form. The function **lastlinep**

(the suffix "p" is traditional Lisp nomenclature for predicates) tests for the current point being on the last line of the buffer. The function **display-error** causes an error message to be printed at the bottom of the screen (via primitives intrinsic to the redisplay: Lisp I/O may not be used in the extension environment), and a non-local transfer of control out of the whole function (**find-first-blank-line**). It is for this reason that a **stop-doing** is not needed after the call to **display-error**.

Experience with the extension language has shown that its meaning is so transparent that the underlying Lisp is all but invisible: the emphasis of Lisp shifts from its data world to its being a formalism for organizing function invocation.

---



*bsg@basistech.com (author)*
*thvv@best.com (site collator/maintainer)*


Multics home page

Features

History

Sites

Multicians

Glossary

Stories

Biblio-graphy

Multics general